

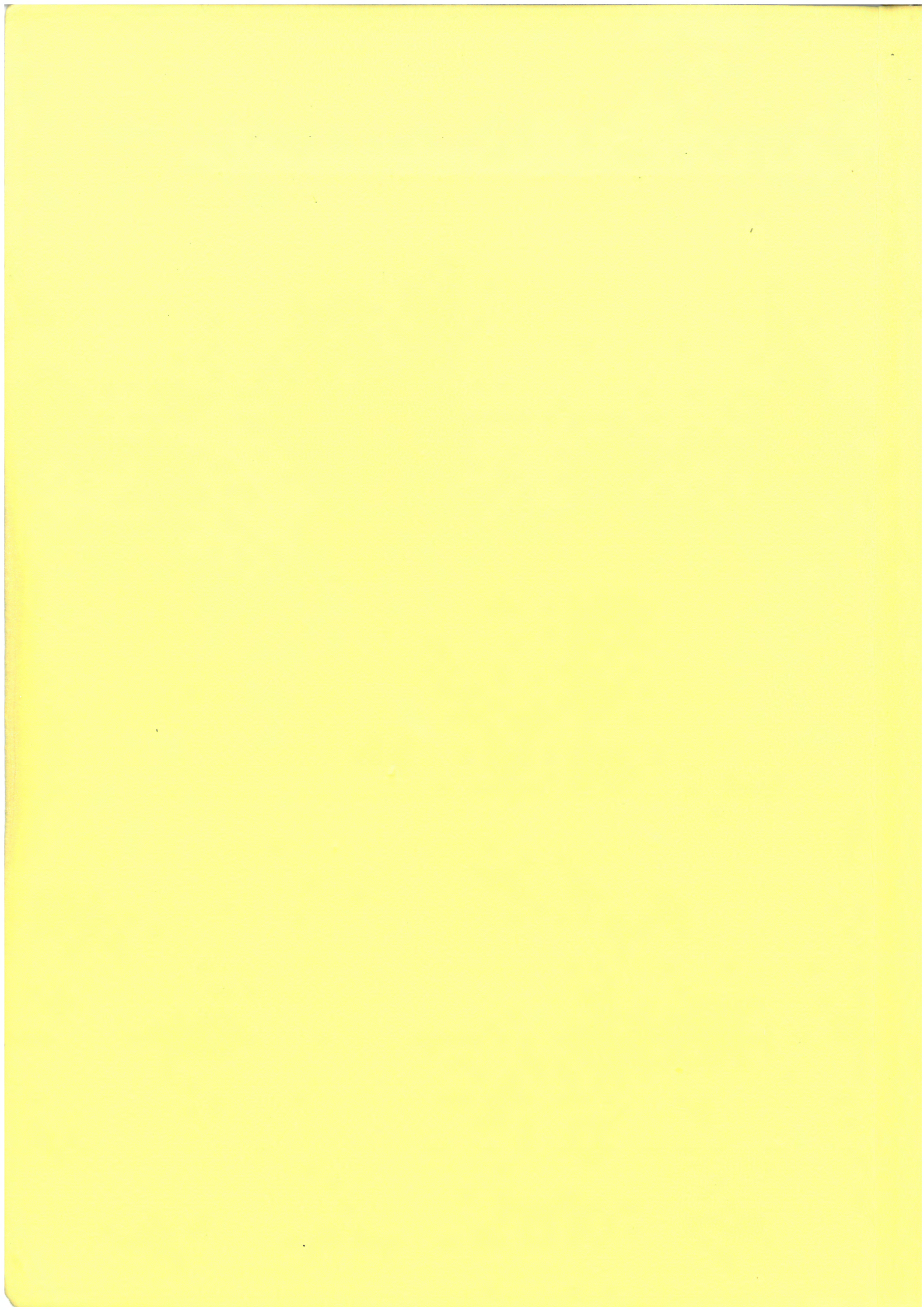
INSIDE A COMPUTER

by

M H LINCK

Copyright c 1992
M H Linck

All rights reserved
March 1992 v1



PREFACE

This text sets out to cover, in concept, how a computer works. The principles used are explained and built on to show this. A simple assembler, **SMALL**, is introduced. Finally simple Input/Output is expanded upon and the concept of micro-programming introduced.

I wish to thank my colleagues Donald Cook & Riel Smit for their help & constructive suggestions. Every effort has been made to ensure that the text and the programs are correct. Any errors that remain are my responsibility. I would appreciate being informed of them.

M H Linck

INSIDE A COMPUTER

INDEX

CHAPTER	TITLE	PAGE NUMBER
1	History	1
2	Number Systems	7
3	Boolean Algebra & Circuits	13
4	Number Representation inside the computer	25
5	Hardware Implementation	49
6	The Computer & the Assembler	61
7	Loops in Assembler	79
8	Address Modification	85
9	Procedures	100
10 X	Characters	109
11 >	Machines with differing Numbers of Addresses	116
12	The Structure of an Assembler	123
13 >	Input/Output concepts	137
14 >	Control Unit Structures	149

✓
134, 135, 136
149 150

CHAPTER 1

HISTORICAL INTRODUCTION TO COMPUTERS

INTRODUCTION

In this chapter I will first give a short history of computing devices and then contrast the 5 generations of electronic computers. Finally I will discuss the structure of a simple electronic computer

A SHORT HISTORY OF COMPUTING

VERY EARLY DAYS

The FINGERS of the hand must be the first device to help a human count. HEAPS of stones, to represent numbers, were the next tool. Stonehenge, 1600BC, is an example.

From about 1000BC the ABACUS has been used primarily for addition & subtraction. This device is still in use in the East today. (In a competition in the mid-1940's skilled operators added faster on an abacus than their opponents on an electro-mechanical calculator).

In the 1600's the principle of LOGARITHMS was understood (ie multiplication by addition of logarithms & division by subtraction). This led to the invention of the SLIDE-RULE.

In 1642 PASCAL invented a machine to add & subtract. In the place of the beads of the abacus he used cylinders with the digits 0-9 engraved on the circumference. Carry digits were transmitted to the next column by direct gearing of successive shafts.

In 1694 LEIBNITZ made a machine that could also multiply & divide. Again cog-wheels & shafts were used in its construction.

JACQUARD, in 1801, built a weaving loom that used PUNCHED CARDS to control the movements of the threads. A new card with different holes would alter the pattern.

Using wheels & shafts BABBAGE in 1822 built a DIFFERENCE ENGINE that could be used to calculate the value of a polynomial. Later on he worked on a general purpose calculating machine called the ANALYTICAL ENGINE (1834+) which had many features of current computers. This machine had a store and an arithmetic unit. The operations that the machine had to perform could be fed in manually or from punched cards. He also anticipated a primitive printer for output. The distinction between his difference engine & the analytic engine is that of a special purpose calculator to general purpose computer. (Lady ADA Lovelace

suggested cards could be prepared to instruct Babbage's engine to repeat certain operations. Because of this she became known as the first programmer. Her name has been given to one of the newest computer languages.)

In the late 1800's HOLLERITH designed a machine that could, by electrical means detect the presence/absence of a hole in a card. The US census in 1890 was done using this machine. Essentially it was a Sorting machine. POWERS used this idea to develop accounting machines. The data was punched on cards. The program, however, was specified by wiring a plug-board attached to the computer. When a new program was to be run this board had to be re-wired.

J Atanasoff & C Berry (1937-1942) invented the first electronic digital computer. It was called ABC.

In 1939-44 AIKEN, with IBM, constructed an automatic sequence controlled calculator, the Harvard Mark1. Paper tape was used to input both the data and the program. Counter wheels were used to store numbers in decimal form. Speedwise it could add, multiply & divide in .3, 4 & 10 secs respectively.

REAL COMPUTERS 1946

ENIAC (Electronic Numerical Integrator and Calculator) was built in 1946 & used 18000 valves. It could only store 20 numbers but could add, multiply & divide in .0002, .003 & .006 sec.(J Presper Eckert & Mauchly)

TURING developed a mathematical theory of an automatic computing device. He devised a method to determine the minimum number of steps that a machine needs to solve a problem. He also devised a theoretical method for deciding if a problem is soluble or not.

VAN NEUMAN (1945) devised the concept of an INTERNALLY-STORED program for a computer. This is the heart of present day computer design.

Using these concepts the first two computers, as we know them today, were developed and completed in 1949.

EDVAC (Electronic Discrete Variable Automatic Computer) was built in the US by von Neuman, J Presper Eckert & Mauchly.

EDSAC (Electronic Delay Storage Automatic Calculator) was built in Cambridge, England by M Wilkes.

Both these machines were electronic and used the internally-stored program concept.

The Americans realised the commercial potential of the computer faster than the British. Remington-Rand produced UNIVAC in 1951, and IBM entered the market in 1953 with its IBM 701 machine. In 1959 it introduced two new machines the IBM 1401 and the IBM 1620 & the IBM System 360 in 1964 (which was a range of compatible machines). The British were slower developing a LEO commercial machine. The predominant British company is ICL (previously ICT).

From these first machines to those of today many changes have taken place. In the **HARDWARE** sphere pretty well everything is **FASTER**, **SMALLER** and **MORE RELIABLE**. Certainly new technologies have evolved but the basic concepts have remained the same. In the next section the five generations of computer systems are discussed.

On the **SOFTWARE** side as great or greater changes have been seen. On the first computers the instructions had to be given in binary code (ie 10011010 which might mean **ADD**). The next advance was to write in Assembler language ie **ADD 45**. In 1957 the first High-level language, **FORTRAN**, was developed. It was a Scientific language & you could now write expressions like $sum = a + b$. The Commercial Data Processing language **COBOL** followed in 1959 and many other languages thereafter. To use these high-level languages a **COMPILER** is necessary to convert the sophisticated statements in that language to binary instructions that the computer can understand. More on this subject later.

To date five generations of computer systems have been recognised. Each generation has certain software and hardware characteristics specific to it.

THE 5 COMPUTER GENERATIONS

In this section the 5 major generations of computer systems are described. The main attributes of each generation are discussed under the headings of Hardware, Software & Computer Organisation. Naturally there is some blurring across the boundaries from one Generation to another.

1ST GENERATION (1946-1959)

HARDWARE: Vacuum tubes

SOFTWARE: Assembly Language.

ORGANISATION: Every instruction done strictly in sequence.

2ND GENERATION (1959-1964)

HARDWARE: Transistors, Magnetic Core Memory.

SOFTWARE: High-level Languages (Fortran, Cobol)
Input/Output Control Systems (A very simple Operating System allowing a sequence of programs to be run automatically one after another.)

ORGANISATION: Input/Output Operations could be overlapped with calculations by use of interrupts.

3RD GENERATION (1964-1972)

HARDWARE: Integrated circuits.

SOFTWARE: Operating Systems. Multiprogramming,
Time-Sharing.

ORGANISATION: Memory could be split between several
users at the same time.

4TH GENERATION (1972- ~1990)

HARDWARE: Large scale integrated circuits. Mini-
computers, micro-computers. Reductions in
hardware costs.

SOFTWARE: Graphics, Computer-Aided-Instruction,
Databases & Games.

ORGANISATION: Modular design of hardware (easier to
fix). Memory protection. A lot of software
now incorporated into the hardware.
Networks.

5TH GENERATION (~1990 --)

HARDWARE: Even larger faster computers. Possibly
incorporating Parallel Processing & Special
purpose processors.

SOFTWARE: Expert Systems, Artificial Intelligence.

ORGANISATING: Integration of Voice, Video & Computing.

STRUCTURE OF A SIMPLE ELECTRONIC COMPUTER

The basic units of an electronic computer are:

- Input device
- Output device
- Memory
- Central Processing Unit (CPU). Comprising:
 - Arithmetic & Logic Unit
 - Control Unit
 - CPU Registers

The inter-connection of these units is shown in Figure 1.1.

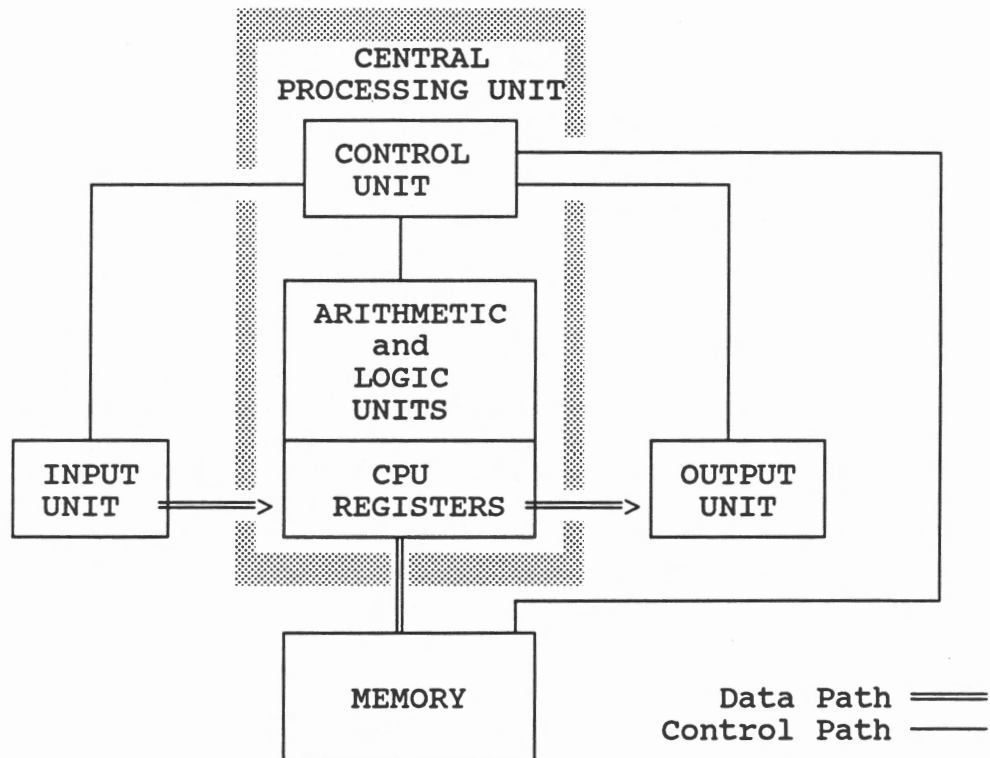


FIGURE 1.1

INPUT UNIT

This is the unit that is used to get data from the outside world into the computer. This data is transmitted to a specific register of the Central Processing Unit (CPU). The keyboard is an example of an input unit.

OUTPUT UNIT

This unit is used to obtain data from the computer in some readable form. The contents of a specific output register of the CPU is sent to the output unit. A printer is an example of an output unit.

MAIN MEMORY

The main memory of a computer is the unit that stores both the **PROGRAM** that is to be executed as well as the **DATA** pertaining to the problem. The main memory is made up of individual words. Each word can hold either a computer instruction or some data value. The contents of a word of memory can be sent to some register in the CPU or vice-versa. The main memory of a common micro-computer might consist of 256×1024 (256K) words each of 16 binary bits.

CENTRAL PROCESSING UNIT

REGISTERS

Several registers exist in the CPU. Most registers will hold some data value. These registers are used in conjunction with the Arithmetic & Logical Unit. One register is specifically used to hold the Current Instruction being obeyed by the computer.

ARITHMETIC AND LOGICAL UNIT

This unit carries out the arithmetic and logical functions that the computer is capable of performing.

In overview the arithmetic functions that can be performed are +, -, *, /. In the case of addition the contents of 2 registers of the CPU are input to the ADD circuit which produces as output the sum of these values. This output is sent to some specific register of the CPU (usually the ACCUMULATOR).

A similar situation exists for the LOGICAL functions like AND, OR, NOR etc.

CONTROL UNIT

This unit controls exactly what the computer is doing at any specific moment in time. In brief, the Control Unit causes an instruction to be loaded from memory to the Current Instruction Register in the CPU. Then the Control Unit causes that instruction to be executed. The Control Unit also controls the action of the Input & Output Units as well as the Memory.

CHAPTER 2

NUMBER SYSTEMS

INTRODUCTION

In this chapter the representation of numbers to any base will be discussed. Specifically numbers base 10, 2, 8 & 16 will be discussed because of their applicability to computing. Conversion between these number systems will be discussed as will the fundamental operations of addition, subtraction, multiplication & division.

REPRESENTATION OF NUMBERS

The human race uses the decimal number system undoubtedly because it has 10 fingers.

Any decimal number can be expressed as an expansion of digits multiplied by the appropriate power of 10 (the base).

$$314.73 = 3 \cdot 10^2 + 1 \cdot 10^1 + 4 \cdot 10^0 + 7 \cdot 10^{-1} + 3 \cdot 10^{-2}$$

Base (or Radix) 10 is not sacred. In the general case any base can be used.

The number

$$A_n A_{n-1} \dots A_1 A_0 . A_{-1} A_{-2} \dots A_{-m} \text{ (base } R)$$

is a short-hand representation of

$$A_n R^n + A_{n-1} R^{n-1} + \dots + A_1 R^1 + A_0 R^0 + A_{-1} R^{-1} + A_{-2} R^{-2} + \dots + A_{-m} R^{-m}$$

For computing systems Base 2 is used because of the binary nature of computer stores.

$$\begin{aligned} 1101.01 &= 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} \\ &= 8 + 4 + 0 + 1 + 0 + 0.25 \\ &= (13.25)_{10} \end{aligned}$$

For any large decimal number its binary equivalent will be inconveniently LONG. It has become conventional in such cases to express the number in the octal system (base 8) or in hexadecimal (base 16) because the number is a lot shorter in either of these systems. For example:

$$(1378)_{10} = (10101100010)_2 = (2542)_8 = (562)_{16}$$

4 digits 11 digits 4 digits 3 digits

CONVERSION BETWEEN NUMBERS IN DIFFERENT BASES

BASE 10 -- BASE 2

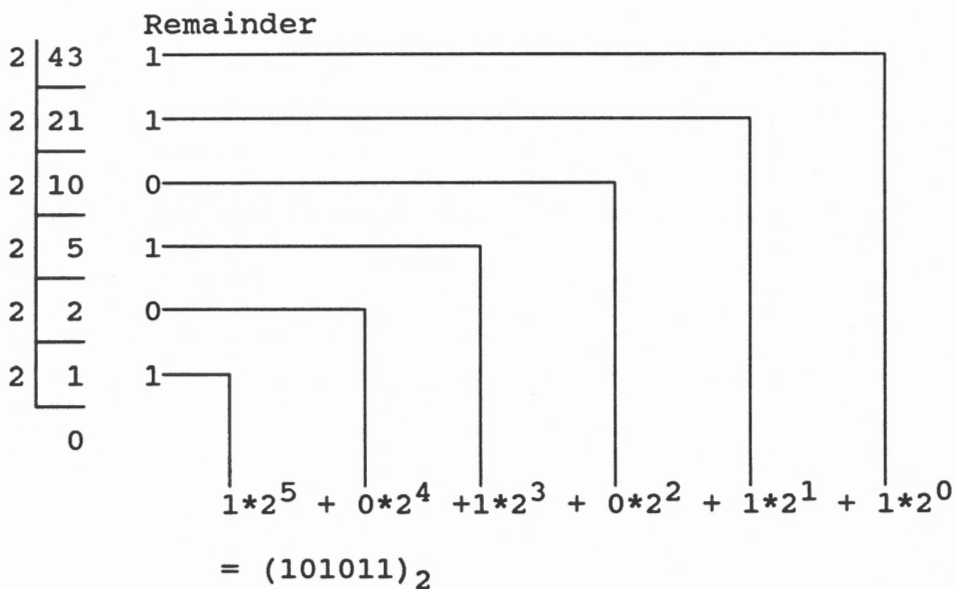
The integer and fractional parts of the number are handled separately.

For the integer part successively divide the number by 2. The remainder is the least significant digit of the result. Stop when there is no remainder.

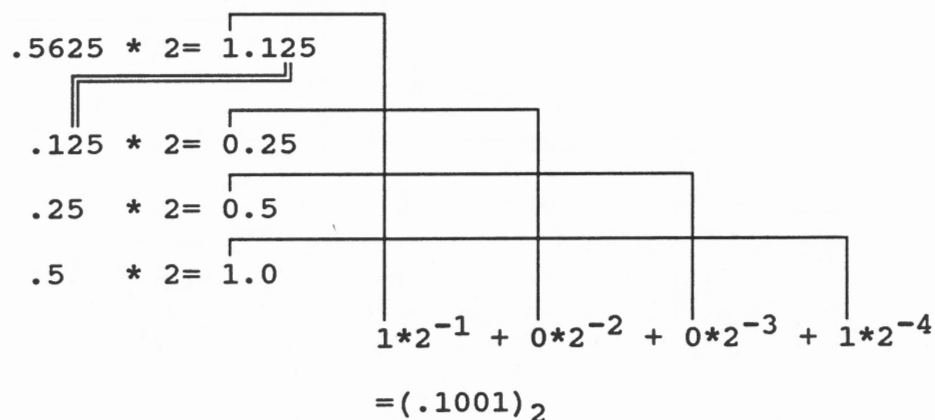
For the fractional part multiply the number by 2. The whole number part of the result is the next digit of the result. Stop when there is no remainder.

Example: Convert $(43.5625)_{10}$ to binary (base 2).

Integer part:



Fractional part:



Thus $(43.5625)_{10} = (10011.1001)_2$

BASE 2 -- BASE 8 (OCTAL)

The digits of the octal system are: 0,1,2,3,4,5,6,7.

METHOD: Starting at the radix point group the binary digits in groups of 3 (Add leading and trailing zeros, as necessary, to make up a group of three digits). Express each group of 3 binary digits as an octal number.

Examples:

$$\begin{aligned}
 (43.5625)_{10} &= (101011.1001)_2 \\
 &= (101011.100100)_2 \quad \text{Two trailing zeros added} \\
 &= (101 \ 011.100 \ 100)_2 \\
 &= (5 \ 3 . 4 \ 4)_8 \\
 &= (53.44)_8
 \end{aligned}$$

$$\begin{aligned}
 (23.5)_{10} &= (10111.1)_2 \\
 &= (010 \ 111.100)_2 \\
 &= (2 \ 7 . 4)_8 \\
 &= (27.4)_8
 \end{aligned}$$

An alternative method is to successively divide the integer part of the number to be converted by 8. The remainder, at each step, becomes the least significant digit of the result. For the fractional part multiply by 8. The whole number part is the next digit of fractional part of the answer.

BASE 2 -- BASE 16 (HEXADECIMAL)

The digits of the hexadecimal system are: 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F.

METHOD: Starting at the radix point group the binary digits in groups of 4 (Add leading and trailing zeros, as necessary, to make up a group of four digits). Express each group of 4 binary digits as a hexadecimal number.

Examples:

$$\begin{aligned}
 (43.5625)_{10} &= (101011.1001)_2 \\
 &= (00101011.1001)_2 \quad \text{Two leading zeros added} \\
 &= (0010 \ 1011.1001)_2 \\
 &= (2 \ B . 9)_{16} \\
 &= (2B.9)_{16}
 \end{aligned}$$

$$\begin{aligned}
 (23.5)_{10} &= (10111.1)_2 \\
 &= (0001 \ 0111.1000)_2 \\
 &= (1 \ 7 . 8)_{16} \\
 &= (17.8)_{16}
 \end{aligned}$$

An alternative method is to successively divide the integer part of the number to be converted by 16. The remainder, at each step, becomes the least significant digit. For the fractional part multiply by 16. The whole number part is the next digit of the fractional part of the answer.

BASE 2,8,16 TO BASE 10

Use the definition.

Examples:

$$\begin{aligned}(1101.01)_2 &= 1*2^3 + 1*2^2 + 0*2^1 + 1*2^0 + 0*2^{-1} + 1*2^{-2} \\ &= 8 + 4 + 0 + 1 + .5 + .25 \\ &= (13.75)_{10}\end{aligned}$$

$$\begin{aligned}(721.34)_8 &= 7*8^2 + 2*8^1 + 1*8^0 + 3*8^{-1} + 4*8^{-2} \\ &= 7*64 + 2*8 + 1*1 + 3/8 + 4/64 \\ &\quad 448 + 16 + 1 + .375 + .0625 \\ &= (465.4375)_{10}\end{aligned}$$

$$\begin{aligned}(B3.2F)_{16} &= B*16^1 + 3*16^0 + 2*16^{-1} + F*16^{-2} \\ &= 11*16^1 + 3*16^0 + 2*16^{-1} + 15*16^{-2} \\ &= 176 + 3 + 2/16 + 15/256 \\ &= (189.1836)_{10}\end{aligned}$$

OPERATIONS ON NUMBERS

In the decimal system addition, subtraction, multiplication & division are well known and understood. These operations work in exactly the same way for any number system irrespective of the base of the system. Do remember that the carry from one column to the next is the value of the BASE of the system that you are working in. Some examples are given in Figure 2.1. The values of the powers of 2, 8 & 16 is given below in Table 2.1.

$2^0 = 1$	$8^0 = 1$	$16^0 = 1$
$2^1 = 2$	$8^1 = 8$	$16^1 = 16$
$2^2 = 4$	$8^2 = 64$	$16^2 = 256$
$2^3 = 8$	$8^3 = 512$	$16^3 = 4096$
$2^4 = 16$	$8^4 = 4096$	$16^4 = 65536$
$2^5 = 32$	$8^5 = 32768$	
$2^6 = 64$		
$2^7 = 128$		
$2^8 = 256$		
$2^9 = 512$		
$2^{10} = 1024$		

TABLE 2.1 -- POWERS OF 2, 8 & 16

Base 10	Base 2	Base 8	Base 16
<u>ADDITION</u>			
34 +15 --- 49	100010 +001111 ----- 110001	42 +17 --- 61	22 F --- 31
$4 \times 10 + 9 = 49$	$1 \times 32 + 1 \times 16 + 1 = 49$	$6 \times 8 + 1 = 49$	$3 \times 16 + 1 = 49$
<u>SUBTRACTION</u>			
34 -15 --- 19	100010 -001111 ----- 010011	42 -17 --- 23	22 - F --- 13
	$1 \times 16 + 1 \times 2 + 1 = 19$	$2 \times 8 + 3 = 19$	$1 \times 16 + 3 = 19$
<u>MULTIPLICATION</u>			
34 18 --- 272 340 --- 612	100010 10010 ----- 000000 100010 ----- 000000 100010 ----- 1001100100 512+128+64+4 =612	42 22 --- 104 1040 ----- 1144 1*512 +1*64 +4*8 +4 =612	22 12 --- 44 220 --- 264 2*256 +6*16 +4 =612
<u>DIVISION</u> 306/17=18			
18 17 306 17 --- 136 136 --- 000	10010 10001 100110010 10001 ----- 10001 10001 ----- 0	22 21 462 42 -- 42 42 -- 00	12 11 132 11 -- 22 22 -- 00

FIGURE 2.1

EXAMPLES

1 Convert the following decimal numbers to binary:

(a) 13 (b) 43 (c) 4078 (d) 40780

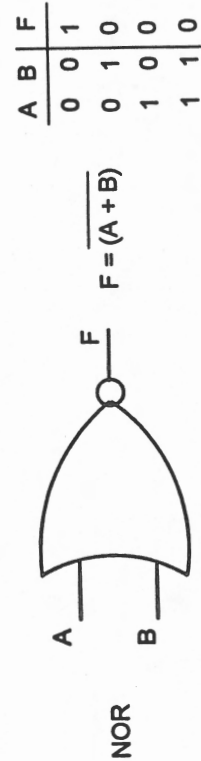
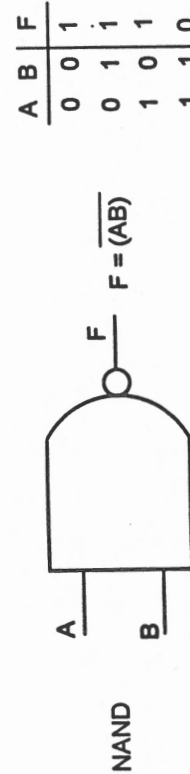
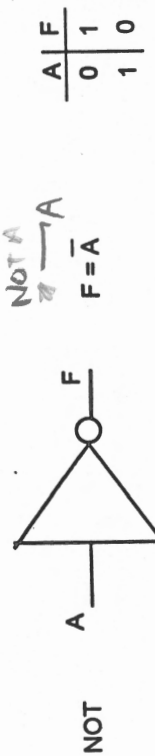
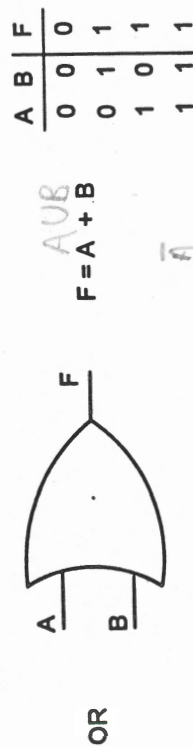
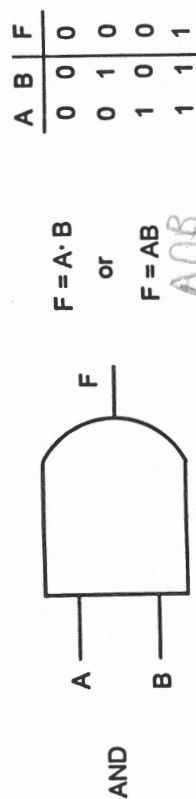
2. Convert the following binary numbers to decimal:

(a) 1101 (b) 1110100 (c) 10001 (d) 11110111

3 Complete the following table:

Decimal	Binary	Octal	Hexadecimal
36			
63	101010 1101101	43 764	2A5 ABD

4 For the bases: 10, 16, 8 & 2 calculate: 19×23 , $234 - 176$, $256 + 389$, $372 / 31$.

LOGIC GATES AND THEIR TRUTH TABLES

CHAPTER 3

BOOLEAN ALGEBRA & CIRCUITS

INTRODUCTION

In this chapter I will consider, IN CONCEPT, how a computer works at the electronic level. The concepts of Boolean Algebra will be explained and will be used to show how a circuit to ADD two binary numbers can be set up. I will also show how the contents of one word (or register) can be moved to another word (or register). Finally an explanation of how the Control Unit operates will be given.

It must be remembered that ALL information is stored in a digital computer in BINARY form (ie 5 = 101). Hence all the operations of a digital computer deal with Binary digits. Thus the laws of Boolean Algebra can be used in the design of all circuitry of an electronic digital computer.

BOOLEAN ALGEBRA

Boole (1815-1864) discovered and refined the 2-state algebra that has been named after him. At the time of discovery there was no use for this algebra and it was only after Claude Shannon, in 1938, discovered the isomorphism between Boolean Algebra and telephone switching circuits that it came to prominence. From there it was a small step to use this algebra in the design of computer circuits. Boolean algebra can be used to describe what a circuit is to do. Its laws can also be used to design the cheapest circuit to perform that operation.

BOOLEAN OPERATORS

There are 3 primary Boolean Operations: AND, OR & NOT. They are defined as follows:

AND (· IS AN ALTERNATE SYMBOL FOR AND).

The Truth table for AND is given in Figure 3.1.

Input		Output
A	B	A . B
0	0	0
0	1	0
1	0	0
1	1	1

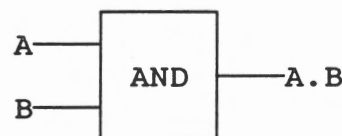
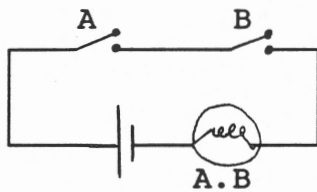


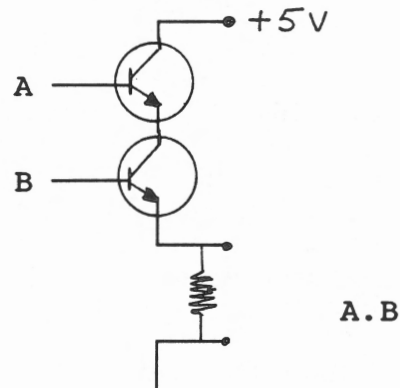
FIGURE 3.1

My computer is happy with NAND and NOR but
can't handle BILL gates!

The AND operator only gives an output when BOTH the inputs are 1. The switching circuit analogy is of a circuit with two switches in series. Both have to be closed for current to flow. Simplistically the switches can be replaced by transistors. Only when both the transistors are conducting is there an output from the circuit. See Figure 3.2



Two switches
in series



Two transistors
in series

FIGURE 3.2

OR

(The '+' is an alternate symbol for OR). The Truth table for OR is given in Figure 3.3.

Input		Output
A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	1

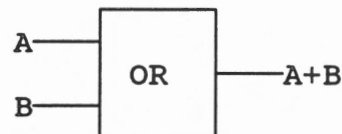
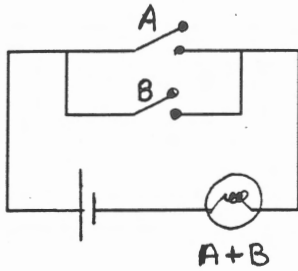


FIGURE 3.3

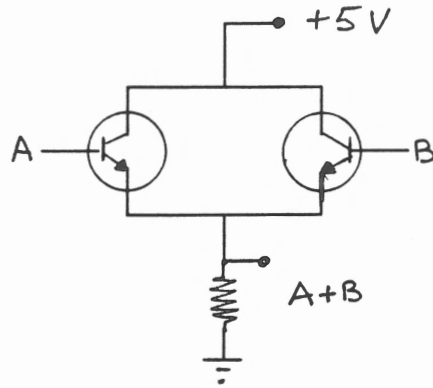
The OR operator gives an output when either or BOTH of the inputs are 1. The switching circuit analogy is of a circuit with two switches in parallel. If either or both are closed the current will flow. Again the switches can be replaced by transistors in the electrical analogue. When either, or both, the transistors are conducting is there an output from the circuit. See Figure 3.4.

Laws

Commutative : $A+B = B+A$, $A.B = B.A$
See page 21



Two switches
in parallel



Two transistors
in parallel

FIGURE 3.4

NOT

The NOT is also known as the COMPLEMENT operator. The truth table is shown in Figure 3.5

Input	Output
A	\bar{A}
1	0
0	1

FIGURE 3.5

This operation changes (or flips) the value of a boolean variable. Symbolically this is represented as shown in Figure 3.6.

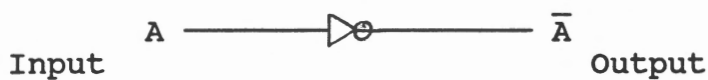
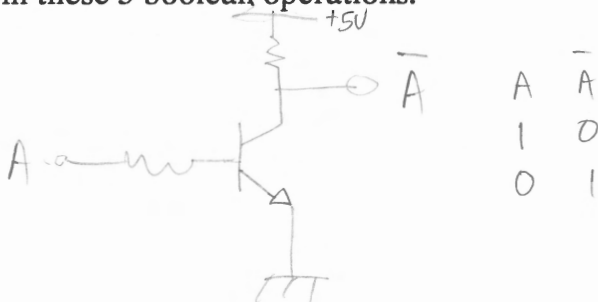


FIGURE 3.6

A simple electronic circuit exists that performs this complement. The details are not given.

All the arithmetic & logical functions available in the a computer can be made up from these 3 boolean operations.



A SIMPLE ADD CIRCUIT (THAT'S INADEQUATE & WRONG)

We wish to ADD two numbers together inside the computer. Let us consider how this might be done. By way of introduction lets add $2+3$ and try by using an AND circuit to do this. This is illustrated in Figure 3.7

2	010	010
+ 3	+ 011	AND 011
--	---	---
5	101	011
(a)	(b)	(c)

FIGURE 3.7

In Figure 3.7(c) the bits of the numbers to be ADDED have been ANDED together starting at the least significant digit. One can see that part of the answer is correct but other parts are wrong. There was no provision for a carry bit. In addition $1 \text{ AND } 1$ gives 1 (not the 0 required). The conclusion is that the AND by itself is not adequate for the job and a more complex circuit is necessary.

HALF ADDER

Lets consider the Truth table for ADDITION, it is shown in Figure 3.8. A boolean circuit to perform this operation is then derived. This is called a HALF ADDER because 2-bits are added together. For full addition a third bit (the CARRY bit) also has to be included. It is worth considering the half adder because it is simple. Also a full adder can be designed using 2 half adders. This will be shown later.

Truth Table for HALF ADDER

Input		Output	
A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

FIGURE 3.8

CARRY CIRCUIT

Lets start with the Carry circuit because it is is simplest. It is only 1 when both A AND B are 1. A simple AND circuit can thus be used and is shown in Figure 3.9.

$$\text{CARRY} = A.B = A \text{ AND } B$$

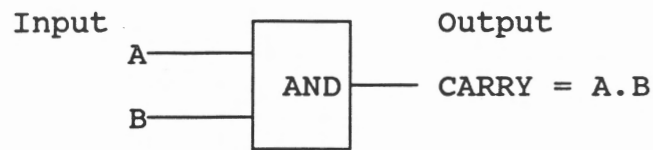


FIGURE 3.9

SUM CIRCUIT

The SUM is true if $\{(A=0 \text{ AND } B=1) \text{ OR } (A=1 \text{ AND } B=0)\}$. Hence

$$\text{SUM} = \bar{A}.B + A.\bar{B}$$

The circuit for the SUM circuit is given in Figure 3.10. It is made up of several of the basic boolean operations.

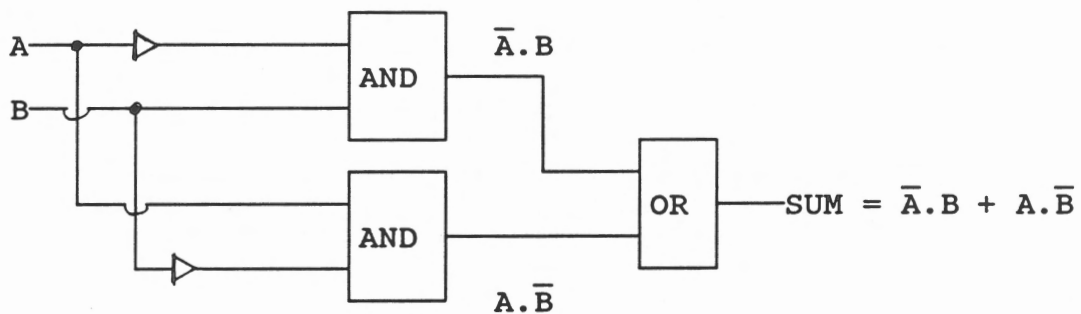
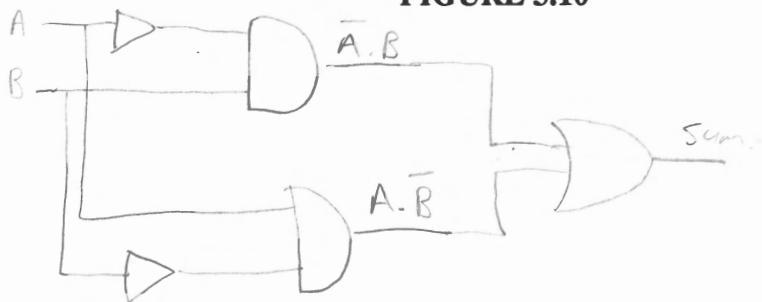


FIGURE 3.10



In practice however we don't just add A and B but also the carry bit from the previous addition. see table over

A	B	XOR	\bar{XOR}
0	0	0	1
0	1	1	0
1	0	1	0
1	1	0	1

$$A \oplus B = \bar{A} \cdot B + A \cdot \bar{B}$$

XOR =

19

$$\overline{A \oplus B} = \overline{A \cdot B} + \overline{A \cdot \bar{B}}$$

$$\text{Simplified SUM} = \bar{A} \cdot (\bar{B} \cdot C + B \cdot \bar{C}) + A \cdot (\bar{B} \cdot \bar{C} + B \cdot C)$$

$$\text{XOR: } A \oplus B$$

look at these paths

$$\bar{A} (B \oplus C) + A (\overline{B \oplus C})$$

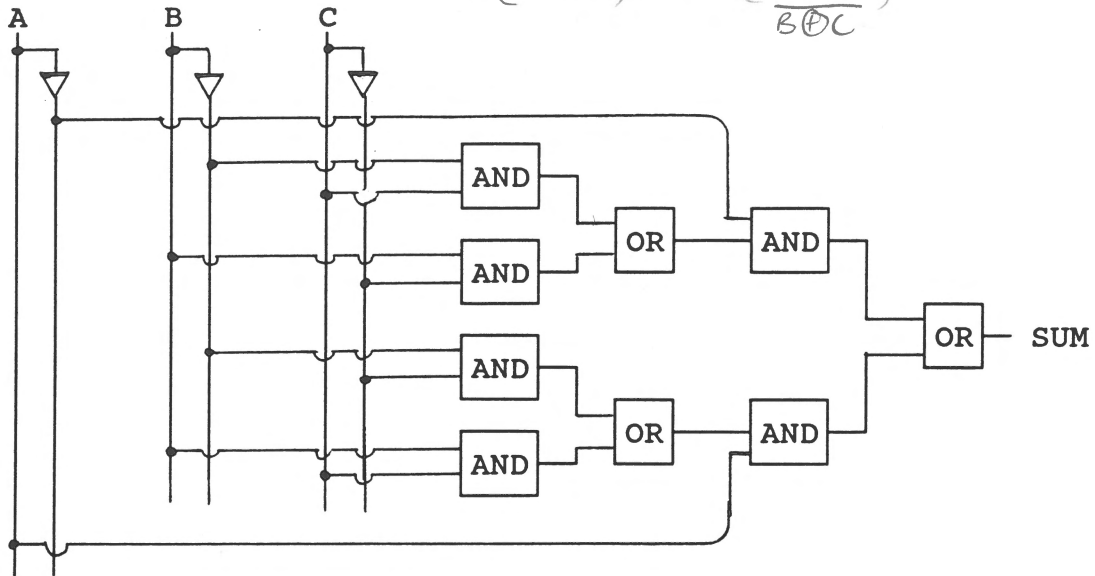


FIGURE 3.12(B)

A FULL ADDER comprising 2 half adders.

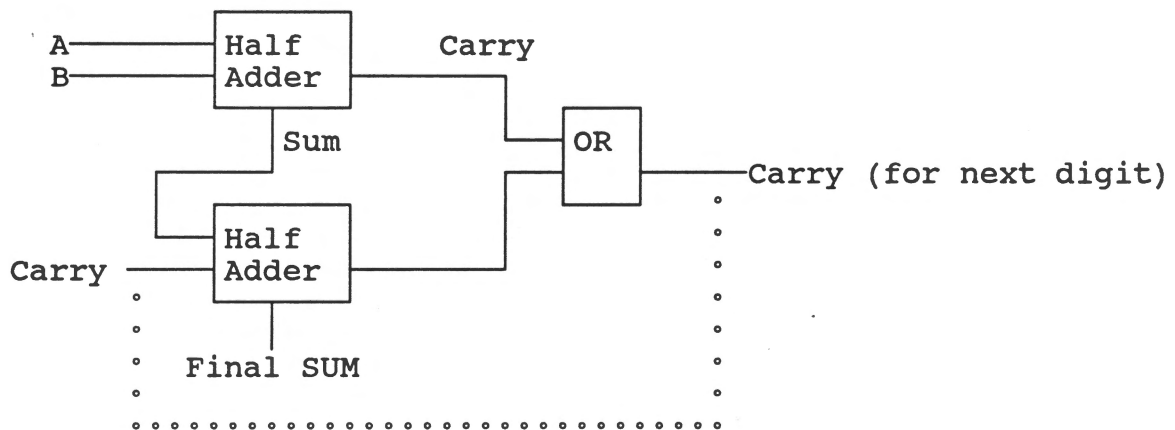


FIGURE 3.13

SERIAL BINARY ADDER

A Serial Binary Adder will perform the addition starting at the least-significant pair of bits and working through to the most-significant pair of bits. The carry is delayed one cycle. The total speed of addition will be the speed that the ADD circuit works at multiplied by the number of bits in the word. The serial addition of bits of a word is shown in Figure 3.14

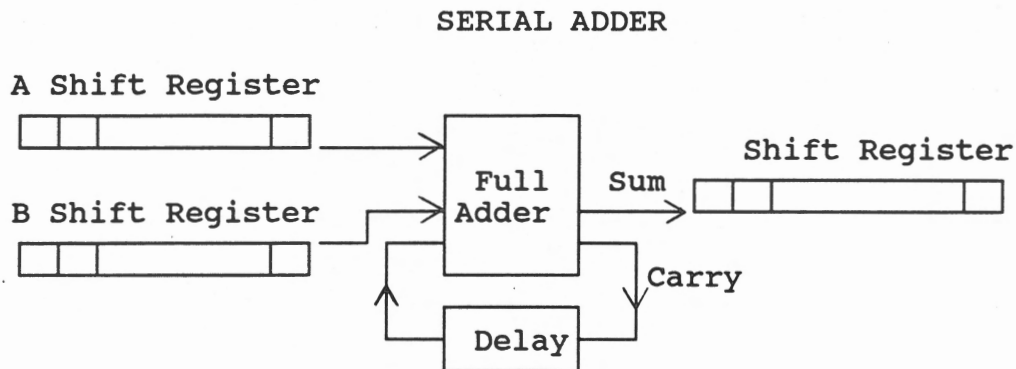


FIGURE 3.14

A shift register has the characteristic that the right-most bit is shifted out (in this case to the Adder), and is also shifted into the left-most bit while all the bits of the word are shifted right 1 bit.

PARALLEL BINARY ADDER

The parallel binary adder adds all the pairs of bits together simultaneously. That's fine but the carry bit from the previous bit position still has to be waited for. This slows the parallel adder down. The parallel adder is faster than a serial adder but not n times faster. Diagrammatically this is shown in Figure 3.15

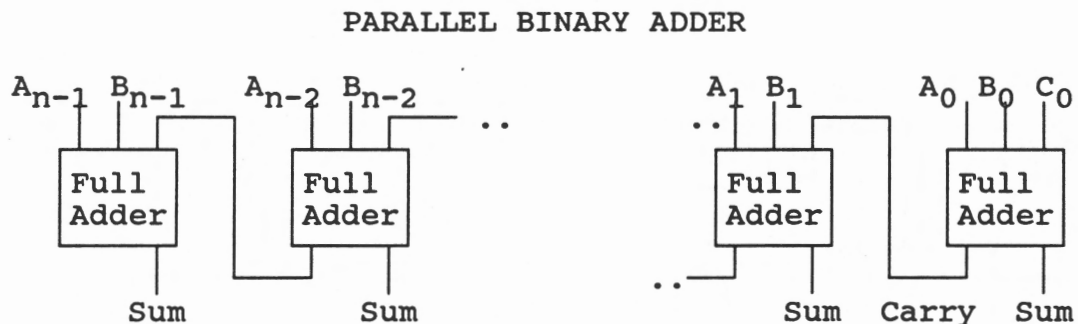


FIGURE 3.15

SET THEORY AND ITS APPLICATION TO BOOLEAN CIRCUITS

The laws of set theory can be used to reduce a boolean expression to its simplest form. The basic laws are given below.

Commutative $A+B = B+A$

Associative $A+(B+C) = (A+B)+C$

Distributive $A+(B.C) = (A+B).(A+C)$

Tautology $A+A = A$

$A.B = B.A$

$A.(B.C) = (A.B).C$

$A.(B+C) = (A.B)+(A.C)$

$A.A = A$

$A.\bar{A} = \text{Null set}$ *empty or*

$A+\bar{A} = \text{Universal set} = 1$

de Morgan $(\overline{A.B}) = \bar{A}+\bar{B}$

$(\overline{A+B}) = \bar{A}.\bar{B}$

$$A.(B+C) = A.B + A.C$$

*Cut a line,
change a sign.*

$$(A'+B') = (A.B)'$$

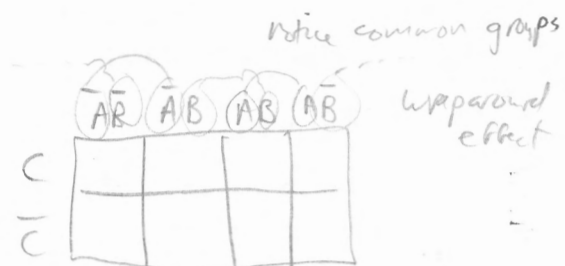
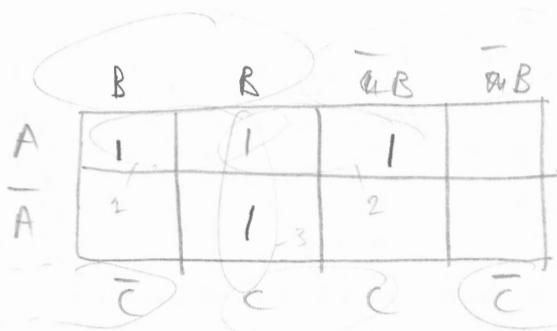
When one is faced with reducing an expression one has to decide which terms to combine to get a reduction. Without going into detail a Karnaugh map is a powerful technique that provides just the information that is needed to make that decision.

NAND & NOR

Two further boolean functions NAND and NOR are used to build circuits. Their truth tables are given in Figure 3.16. These functions are used in practice because it is more convenient to build NAND and NOR circuits. Also some circuits can be built using less components when NAND & NOR are used.

NAND			NOR		
Input		Output	Input		Output
A	B		A	B	
0	0	1	0	0	1
0	1	1	0	1	0
1	0	1	1	0	0
1	1	0	1	1	0

FIGURE 3.16



notice common groups
wrapped effect
notice grouping of all terms

Karnaugh diagram: look at it for carry circuit. Holds true when...
We make groups according to which terms don't matter, i.e. as in expression above, for group 2, the value is true no matter what B is ($B+\bar{B}$).
Similarly, C doesn't matter in 1, and A doesn't matter in 3.

DECODER CIRCUITS

A DECODER is a circuit that uses certain bits of a word to specify the use of one specific electronic circuit from all the circuits possible. In other words it decodes the bits of the word and chooses the appropriate circuit to use to perform the required operation.

INSTRUCTION DECODER

Let us consider the Instruction Register in the CPU. It holds the current instruction to be obeyed and some appropriate memory address. (ie ADD 45. This would mean ADD the contents of word 45 to the accumulator). The Operation code in the Instruction Register must be used to select the ADD circuit for use and to disable all the other op. code circuits. The circuit is shown in Figure 3.17. A 3-bit op code is used.

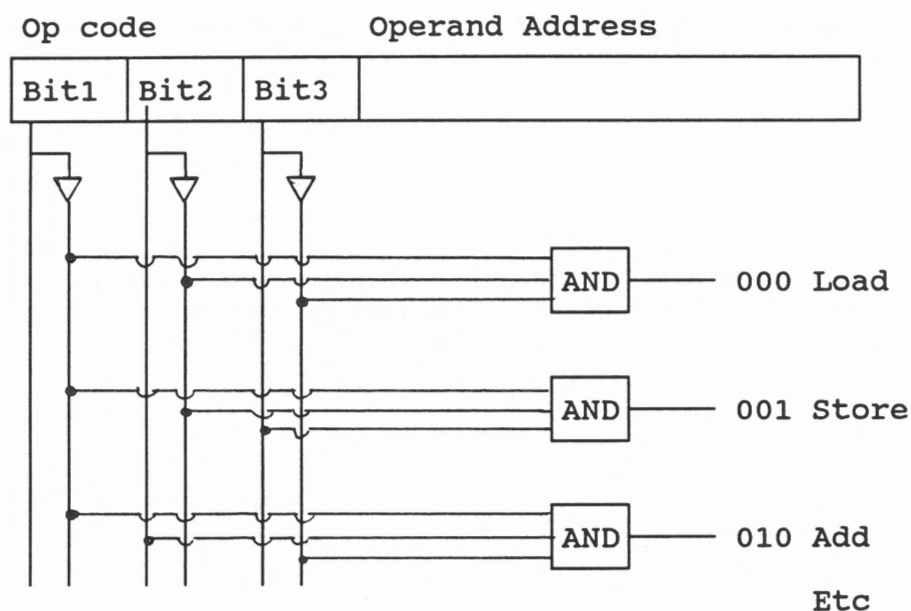


FIGURE 3.17

If the three bits are 000 the ONLY circuit that will be enabled is the ADD circuit. No other AND gate will receive 3 inputs simultaneously thus they will not transmit any signal

ADDRESS DECODER

The address decoder works in exactly the same fashion. In this case the address part of the instruction is used to specify which word is to be accessed while ALL other words are disabled.

CONTROL UNIT

The Control Unit is the heart of the computer. The control unit causes the next instruction to be loaded from memory to the instruction register. Then it causes that instruction to be executed. This sequence is repeated for ever. Diagrammatically this is shown in Figure 3.18

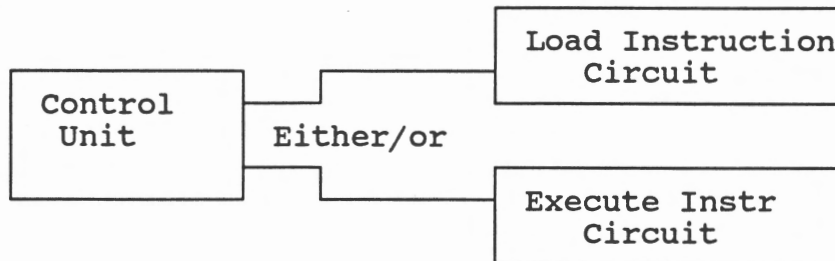


FIGURE 3.18

Electronically the Control Unit is a circuit that, in sequence, sends a signal (or pulse) first to the 'load an instruction' circuit then to the 'execute' circuit.

The 'load an instruction' circuit causes the contents of the next instruction (which is some specific word of memory [its address is kept in a register in the CPU called the "Location counter"]) to be transfered to the instruction register in memory.

The 'execute' circuit causes:

- (a) the instruction to be decoded.
- (b) the address part of the instruction to be decoded.
- (c) the instruction to be carried out.

The Control Unit operates at some specific speed. Often this is known as the clock speed of the computer. The faster this speed of the clock the faster the computer works. Of course the pulse that is given out has to last long enough for the circuit concerned to do its job. Thus the speed of the clock and the speed of the circuitry have to be matched.

EXAMPLES:

1 Give a truth table for a majority function for 3 inputs. This function is true if there are 2 or more true inputs. Give the boolean expression for this function and draw the boolean circuit.

2 Give the truth table for the even function. There are 4 inputs. The function is true if 0, 2 or ALL the inputs are true. Give the boolean expression for this function and draw the boolean circuit. If 0 inputs was not considered even what effect would this have?

3 Show, by evaluating all cases, that De Morgans laws are true. ie show that :

$$\overline{AB} = \overline{A} + \overline{B} \quad \text{and} \quad \overline{A+B} = \overline{A} \cdot \overline{B}$$

4 By evaluating all cases determine if the following are true or not:

$$(1) \quad A \cdot B + \overline{C} \cdot A + C \cdot \overline{B} = A \cdot \overline{B} + A \cdot C + \overline{C} \cdot B$$

$$(2) \quad A \cdot B \cdot C = A \cdot B + B \cdot C + A \cdot C$$

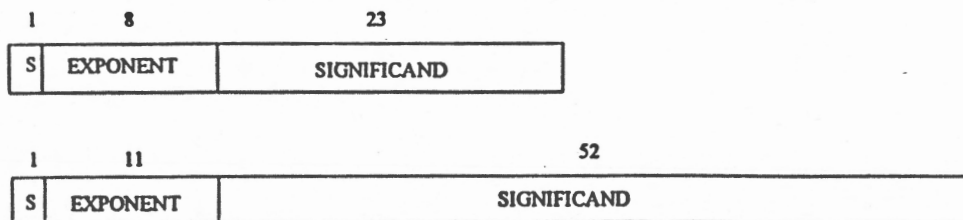


Figure 1: IEEE 754 Floating-Point Number layouts

Values of IEEE 754 Floating-Point Numbers

Single-Precision (32 bits)		
Exponent, e	Significand, f	Value
255	$\neq 0$	Not a Number
255	0	$(-1)^s \infty$
$0 < e < 255$	—	$(-1)^s 2^{e-127} (1.f)$
0	$\neq 0$	$(-1)^s 2^{-126} (.f)$
0	0	$(-1)^s 0$

s = sign bit

Double-Precision (64 bits)		
Exponent, e	Significand, f	Value
2047	$\neq 0$	illegal
2047	0	$(-1)^s \infty$
$0 < e < 2047$	—	$(-1)^s 2^{e-1023} (1.f)$
0	$\neq 0$	$(-1)^s 2^{-1022} (.f)$
0	0	$(-1)^s 0$

CHAPTER 4

REPRESENTATION INSIDE THE COMPUTER

All information has to be represented in some way inside a computer. In this chapter the representation of instructions, characters, integers and reals will be discussed. Also arithmetic inside the computer with integers & reals will be discussed.

BITS, BYTES & WORDS

A bit is the smallest quantity of information that can be stored in a computer. It can either be set (1) or not set (0).

A byte is 8-bits. This is a convenient size in which to represent a character. 256 different combinations can be held in 8-bits.

A word is usually some multiple number of bytes. The exact size of a word is dependent on the specific computer being used. A micro usually has a 16-bit word. Mini-computers often have 32-bit words and mainframe computers generally have larger word sizes 48-bits & so on.

REPRESENTATION OF INSTRUCTIONS

In essence an instruction has two parts: the operation code, and an address. For example: Load the contents of word 45 of memory to the accumulator might be represented as shown below. (16-bit word. 8-bit instruction, 8-bit address).

Load	45		
(Instr 5)			
<table border="1" style="border-collapse: collapse; width: 100%;"> <tr> <td style="padding: 2px 10px;">00000101</td> <td style="padding: 2px 10px;">00101101</td> </tr> </table>		00000101	00101101
00000101	00101101		

It should be appreciated that to make "sense" of the contents of this word it is necessary to treat the left-hand 8-bits specially as an instruction code and the last 8-bits specially as an address.

eb see dick

REPRESENTATION OF CHARACTERS

The codes to be used to represent each character of the alphabet, each number & each special character have to be defined. Two systems are in use EBCDIC and ASCII. EBCDIC (Extended Binary Coded Decimal Information Interchange) is used by IBM and ASCII pretty well by everybody else. The ASCII code (American Standard Code for Information Interchange) is given here. It is a 7-bit code, allowing $2^7 = 128$ different characters, 96 are normal printing characters & 32 are 'non-printing' characters. The bit pattern of each ASCII character is given in Table 4.1 below.

	$b_3b_2b_1b_0$	$b_6b_5b_4$	0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0	0000		NUL	DLE	SP	0	@	P	.	p
1	0001		SOH	DC1	!	1	A	Q	a	q
2	0010		STX	DC2	"	2	B	R	b	r
3	0011		ETX	DC3	#	3	C	S	c	s
4	0100		EOT	DC4	\$	4	D	T	d	t
5	0101		ENQ	NAK	%	5	E	U	e	u
6	0110		ACK	SYN	&	6	F	V	f	v
7	0111		BEL	ETB	'	7	G	W	g	w
8	1000		BS	CAN	(8	H	X	h	x
9	1001		HT	EM)	9	I	Y	i	y
A	1010		LF	SUB	*	:	J	Z	j	z
B	1011		VT	ESC	+	;	K	[k	{
C	1100		FF	FS	,	>	L	\	l	
D	1101		CR	GS	-	=	M]	m	}
E	1110		SO	RS	.	>	N	^	n	~
F	1111		SI	US	/	?	O	-	o	DEL
			Control Characters			Printing Characters				

TABLE 4.1

As examples: 'a' = 110 0001 = (61)₁₆ 'J' = 100 1010 = (4A)₁₆

NUMBER SYSTEMS

INTRODUCTION

In this section the representation of numbers to any base is discussed. Conversion between the decimal system and the binary, octal & hexadecimal systems, & vice-versa, is described. The three systems for representing integer numbers, sign & magnitude, 1's complement & 2's complement are described. Arithmetic using these three systems is described and compared. The representation of real numbers is briefly explained.

REPRESENTATION OF INTEGERS

INTRODUCTION

For simplicity I will assume a 4-bit binary word. All values will be limited to this size.

UNSIGNED INTEGERS

The representation of UNSIGNED integers is straightforward. Simply use the binary representation of the number. Naturally ALL numbers are assumed to be positive.

Examples:

0	0000	5	0101	10	1010
1	0001	6	0110	11	1011
2	0010	7	0111	12	1100
3	0011	8	1000	13	1101
4	0100	9	1001	14	1110
				15	1111

Observe that with 4 bits the range of numbers that can be represented is 0 -- 15 ie $0 \text{ -- } 2^4 - 1$.

For an N bit number the range is $0 \text{ -- } 2^N - 1$

SIGNED INTEGERS

In practice both +ve and -ve integers are required. We are faced with the problem of how to represent the negative number because no - sign exists in the binary system (only 0 & 1 exists). There are two approaches:

(1) Sign & Magnitude

(2) Complement arithmetic (Two methods:- 1's complement & 2's complement will be described)

SIGN & MAGNITUDE

1 bit of the binary word, usually the first, is specifically set aside to represent the sign of the number.

0 -- positive 1 -- negative

The integers will have the following representation:

Decimal	4-bit Binary	Decimal	4-bit Binary
+7	0111	-7	1111
+6	0110	-6	1110
+5	0101	-5	1101
+4	0100	-4	1100
+3	0011	-3	1011
+2	0010	-2	1010
+1	0010	-1	1001
+0	0000	-0	1000
	<div style="display: flex; align-items: center;"> <div style="margin-right: 5px;"> </div> <div style="border-left: 1px solid black; border-top: 1px solid black; width: 40px; height: 15px; margin-left: 5px;"></div> </div> <div style="display: flex; justify-content: space-between; width: 100px; margin-top: 5px;"> Sign Magnitude </div>		<div style="display: flex; align-items: center;"> <div style="margin-right: 5px;"> </div> <div style="border-left: 1px solid black; border-top: 1px solid black; width: 40px; height: 15px; margin-left: 5px;"></div> </div> <div style="display: flex; justify-content: space-between; width: 100px; margin-top: 5px;"> Sign </div>

In this representation there are two different representations for zero (0000 & 1000) plus zero and minus zero.

With a 4 bit word values from -7 to +7 can be represented ie values from $-(2^3-1)$ to $+(2^3-1)$

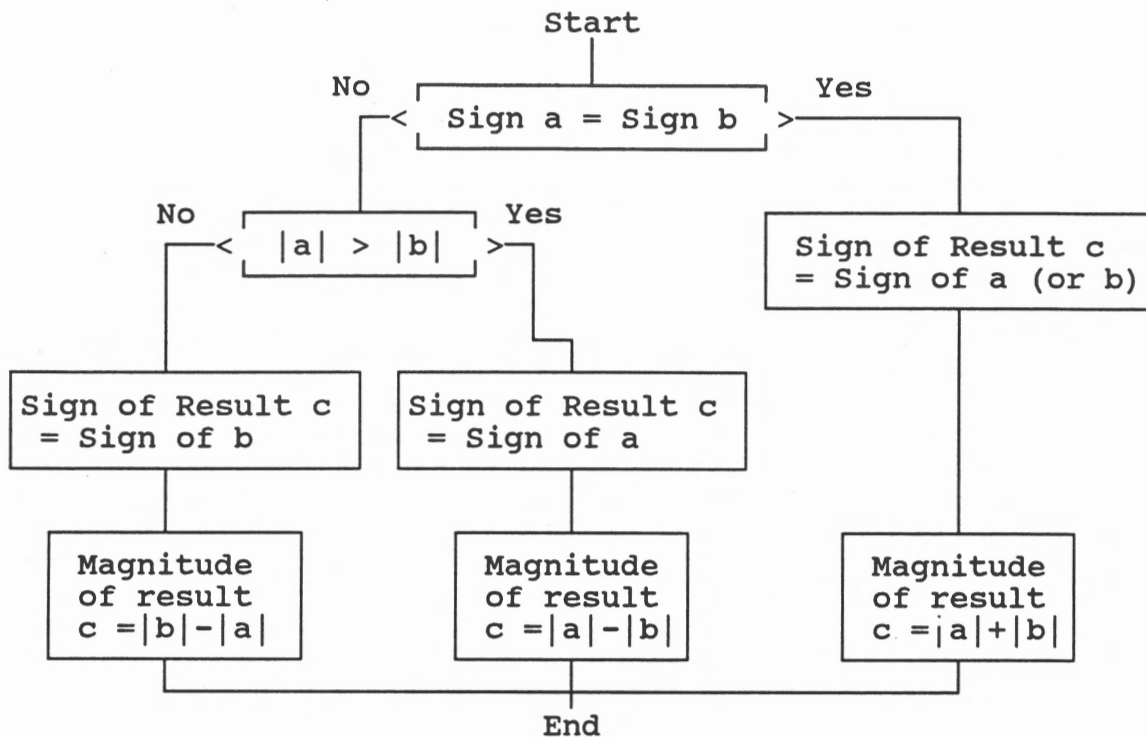
In the general case of an N-bit word the values that can be represented are $-(2^{N-1}-1)$ to $+(2^{N-1}-1)$.

ARITHMETIC

Let us consider how arithmetic is done in this system. It is essential to appreciate that the sign bit and the magnitude bits are treated entirely SEPARATELY.

ALGORITHM FOR SIGN & MAGNITUDE ADDITION $C = (+A) + (+B)$

The algorithm is described in "flowchart" form in Figure 4.2. Thereafter several examples are given.


FIGURE 4.2

Examples:

Numbers	Representation	Result
(+5) + (+2)	0 101 0 010 ----- 0 111	= (+7)
(-5) + (-2)	1 101 1 010 ----- 1 111	= (-7)
(+5) + (-2)	0 101 1 010 ----- 0 011	= (+3)
(-5) + (+2)	1 101 0 010 ----- 1 011	= (-3)

On paper the sign & magnitude system seems a good solution, to the problem. In practice, inside a computer, it is cumbersome to implement. Firstly a special ADD circuit has to be built to implement the entire algorithm given in Figure 3.3. Secondly the sign bit has to be obtained (by shifting out the Left-most bit) then only can the magnitude part of the number be moved out to the adder. There are difficulties here when the difference in magnitude of the two values is the required result.

COMPLEMENT OF A NUMBER

To convert from a positive to a negative number (or vice-versa) simply change the sign bit.

Example:

+5	0101	-5	1101
-3	1011	+3	0011

Note that the magnitude of number remains the same.

ONE'S COMPLEMENT

The following definitions apply:

Positive numbers are represented: $+K = K$
 Negative numbers are represented: $-K = (2^N - 1) - |K|$

Note $N=4$ in our case as we have a 4-bit word. The following numbers can be represented:

Number	1's Complement Representation
7	0111
6	0110
5	0101
4	0100
3	0011
2	0010
1	0001
+0	0000
-0	1111
-1	1110
-2	1101
-3	1100
-4	1011
-5	1010
-6	1001
-7	1000

SOME PROPERTIES OF THIS REPRESENTATION:

COMPLEMENT

To complement a number:

Change every 0 to 1 and every 1 to 0.

Example: $-2 = 1101$ Complement = $0010 = 2$

RANGE OF THE REPRESENTATION

With N bits 2^N states can be represented. So with 4 bits we can represent $2^4 = 16$ different values. In our case:

Largest positive value = 7 or $2^3 - 1$

Largest negative value = -7 or $-2^3 - 1$

In the general case of an N bit word

Largest positive value = $2^{N-1} - 1$

Largest negative value = $-(2^{N-1} - 1)$

ADDITION & SUBTRACTION

BOTH addition & subtraction are done by ADDITION. Any overflow digits are added back into the low order digit position. The addition of this overflow is a straight-forward operation for a digital computer.

Examples of arithmetic are given together with a proof of why arithmetic in this system works correctly is given in Figure 4.3. Four cases only exist & each is dealt with in turn.

Note: $2^N = 2^4 = 10000$

Integer	Representation	Integer	Representation
+5	0101	a	a
+2	0010	b	b
--	----	---	---
+7	0111	a+b	a+b
 $ a > b $			
+5	0101	a	a
-2	1101	-b	$2^{N-1}-b$
--	----	---	-----
+3	1 0010 └─> 1	a-b	$2^N+a-b-1$

	0011 = +3		Overflow bit fed back into low bit. $ a > b $ thus a-b is the correct representation of the result
 $ b \geq a $			
+2	0010	a	a
-5	1010	-b	$2^{N-1}-b$
--	----	---	-----
-3	1100	a-b	$2^N-(b-a)-1$
			Representation of $- b-a $ because $ b > a $
-5	1010	-a	$2^{N-1}-a$
-2	1101	-b	$2^{N-1}-b$
--	----	---	-----
-7	1 0111 └─> 1	-(a+b)	$2^N+[2^{N-1}-(a+b)]-1$

	1000 = -7		Representation of negative number $- a+b $

FIGURE 4.3

MULTIPLICATION

Multiplication is done by successive addition. The correct result can be obtained using the 1's complement system if every carry bit generated is added back into the low order position. In practice this is complicated and thus not done.

Multiplication algorithm:

- 1 Complement any negative value.
- 2 Do normal multiplication.
- 3 Complement the result if either, but not both of the values, were negative.

DIVISION

Division cannot be done directly because:

$$-a/b = (2^N - 1 - a)/b < \{ (2^N - 1 - (a/b)) \} \text{ (Representation of } -a/b)$$

The component $(2^N - 1)/b$ will give rise to non-zero bits in the low order positions & hence give an incorrect answer.

DIVISION ALGORITHM

- 1 Complement any negative value.
- 2 Perform a normal division.
- 3 Complement the result if either, but not both, the numerator and divisor were negative.

Example:

$$-6/3 = -(6/3) = -(0110 / 0011) = -(0010) = 1101 = -2$$

TWO'S COMPLEMENT

The following definitions apply:

Positive numbers are represented:

$$+K = K$$

Negative numbers are represented:

$$-K = 2^N - |K|$$

Note $N=4$ in our case as we have a 4-bit word.

The following numbers can be represented:

Summary of b + c

B	C	b < c	Sum
b	$R^n - c$	$b < c$	$R^n + (b - c)$
b	$R^n - c$	$b \geq 0$	$R^n - (c - b)$

Number	2's Complement Representation
7	0111
6	0110
5	0101
4	0100
3	0011
2	0010
1	0001
0	0000
-1	1111
-2	1110
-3	1101
-4	1100
-5	1011
-6	1010
-7	1001
-8	1000

Provided $|b| < \frac{R^n}{2}$ for
 $R = \text{radix}$ & n the
 number of digits, then
 we represent the number
 b as $R^n - |b|$

$$\begin{array}{r} 5 \quad 0101 \\ -2 \quad 1110 \\ \hline 10 \quad 0011 \end{array}$$

Particular than representing the neg integer as the magnitude (say $-5_{10} = 101_2$) +
 sign, i.e. as 1101_2 , in 2's complement representation we write
 instead $\underbrace{16_{10}}_{R^n} - 5_{10} = 11_{10} = 8 + 2 + 1 = 1011_2$

Summary

Note: $2^N = 2^4 = 10000$

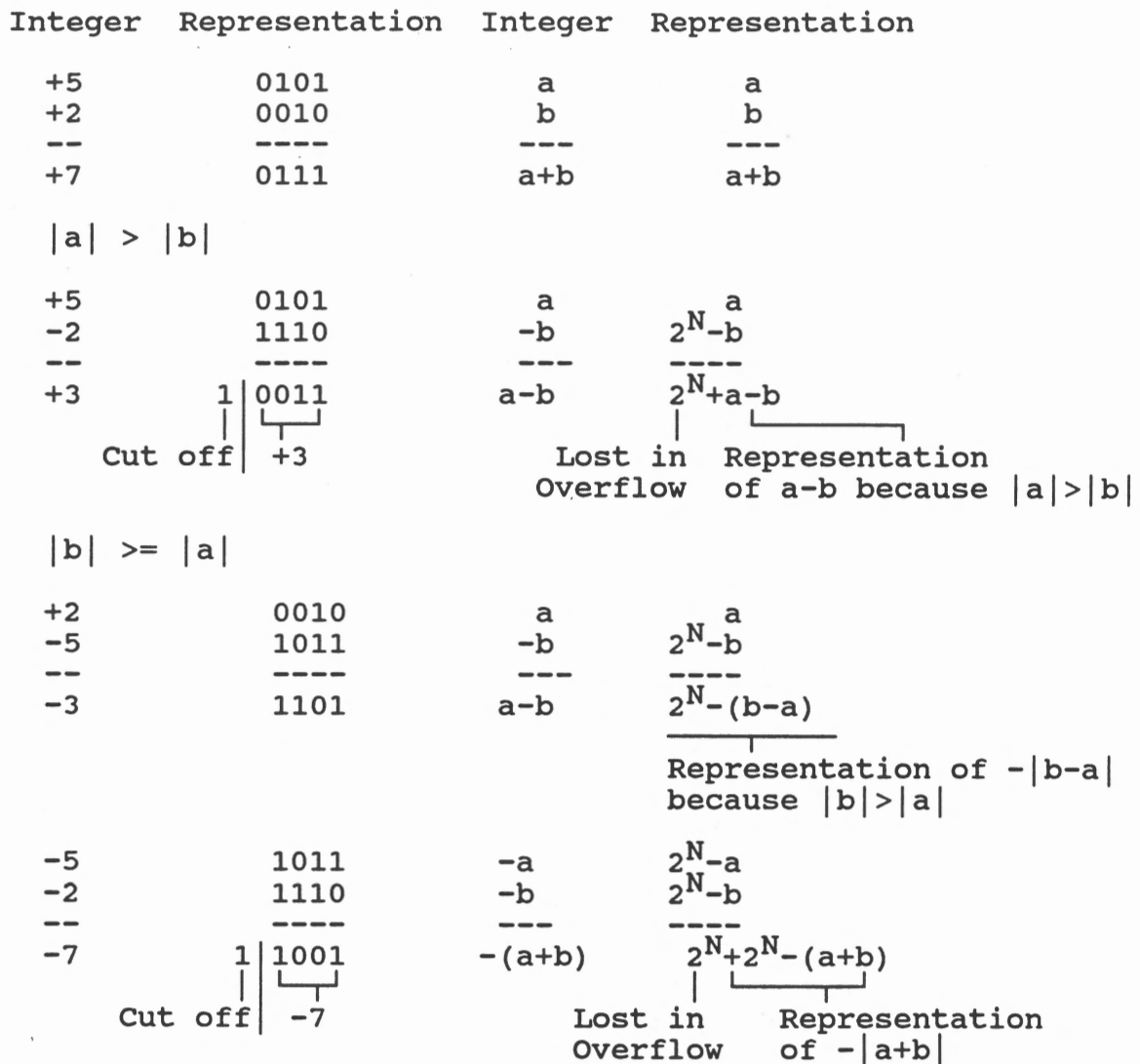


FIGURE 4.4

MULTIPLICATION

Multiplication is done by successive addition. The correct result is obtained using the 2's complement system. This can be proved in the same way as for addition.

DIVISION

Division can not be done directly because:

$$-a/b = (2^N - a)/b \neq (2^N - a/b) = \text{The correct representation of } -a/b$$

ALGORITHM FOR DIVISION

- 1 Complement any negative value.
- 2 Perform a normal division.
- 3 Complement the result if either, but not both, the numerator and divisor were negative.

Example:

$$-6/3 = -(6/3) = -(0110 / 0011) = -(0010) = 1110 = -2$$

COMPARISON BETWEEN SIGN & MAGNITUDE, 2'S & 1'S COMPLEMENT

RULES FOR COMPLEMENTATION

- S&M: Only SIGN bit flipped
 2's: Flip ALL bits and add 1
 1's: Flip ALL bits

RULES FOR ADDITION & SUBTRACTION

- S&M: Special algorithm needed
 2's: Add & Subtract by ADDITION in binary
 1's: Add & Subtract by ADDITION in binary. ADD BACK any overflow bits.

ADVANTAGES & DISADVANTAGES

- S&M: Few advantages for a digital computer.
 2's: Well adapted for serial arithmetic
 Easy detection of overflow
 Simple implementation of Floating point arithmetic
 Addition very simple.
 Complementation more complex than 1's complement.
 1's: Simple complementation
 More complex addition because of end-round carry.
 Inconvenient serial arithmetic because of end-round carry
 Unsatisfactory for detection of overflow.
 Floating point arithmetic more difficult because of end-round carry.

Range (for N bits)		for k a number	
S & M	$-(2^{N-1}-1) \text{ to } (2^{N-1}-1)$	+k	-k
2's	$-2^{N-1} \text{ to } 2^{N-1}-1$	$0 \text{ to } k$	$2^N - k$
1's	$-(2^{N-1}-1) \text{ to } (2^{N-1}-1)$	k	$(2^{N-1}-1) - k$

RANGE (for N bits, radix R)		Representation	
S & M	$-(2^{N-1}-1) \text{ to } (2^{N-1}-1)$	k	-k
2's	$-2^{N-1} \text{ to } 2^{N-1}-1$	2's k	$2^N - k$
1's	$-(2^{N-1}-1) \text{ to } (2^{N-1}-1)$	1's k	$(2^{N-1}-1) - k$

Flip bits

BINARY CODED DECIMAL

An alternative representation of the decimal digits is the BINARY CODED DECIMAL representation. This is given below.

Decimal Digit	BCD Code	
0	0000	
1	0001	
2	0010	The 6 binary codes 1010 thru 1111 are unused (redundant) & have no meaning.
3	0011	
4	0100	
5	0101	
6	0110	
7	0111	
8	1000	
9	1001	

The representation of any number is obtained by representing each individual digit as its 4 bit BCD code. ie

1872 ---> 0001 1000 0111 0010

Arithmetic using this representation is similiar to decimal arithmetic

1872	0001 1000 0111 0010	
+2345	0010 0011 0100 0101	
-----	-----	
4217	0011 1011 1011 0111	
	-1010-1010	Subtract 10 if digit>=10
	0011 0001 0001 0111	
	0001 0001	Add carry bit

	0100 0010 0001 0111	
	4 2 1 7	

While the BCD system seems good it suffers from two disadvantages. Firstly the arithmetic is more complex. The example above illustrates this. Secondly inefficient storage. A BCD digit requires 4 bits. Of the 16 possible states only 10 are used. In spite of these disadvantages BCD is frequently found in applications requiring little storage ie digital watches & pocket calculators.

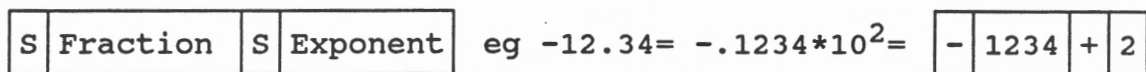
REAL NUMBERS

INTRODUCTION

There are 3 aspects relating to the representation of REAL numbers. They are: Representation, Accuracy & Normalisation.

REPRESENTATION OF REALS

The Real number is broken up into two parts. The FRACTIONAL part (or MANTISSA) and the EXPONENT. Both these parts have a SIGN associated with them.



ACCURACY

The number of bits allocated to the fractional part of the number sets a limit to the accuracy that the real number has.

The range of real numbers that can be held is dependent on the number of bits allocated to the exponent.

NORMALIZATION

To ensure maximum accuracy a real number is normalised. That means that the first digit of the fractional part is not zero. This strategy ensures that unnecessary leading zeros are not stored at the expense of trailing non-zero digits.

REPRESENTATION OF A DECIMAL NUMBER

By way of illustration the representation of a decimal number is shown in Figure 4.5. The decimal point is assumed to be in front of the fraction. The power of the exponent is assumed to be 10.

$$-0.0023589 = -.23589 * 10^{-2} \quad (\text{Normalization})$$

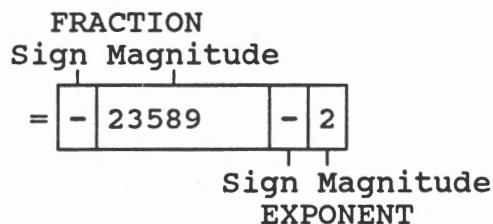


FIGURE 4.5

BINARY REPRESENTATION OF REALS

As described above the fractional part and the exponent are stored separately. The designer of the computer can choose whatever representation he likes for the fractional part and for the exponent. He may choose Sign & Magnitude, 1's Complement or 2's Complement or any combination. Two possibilities will be described. Also the technique of biasing will be demonstrated.

FRACTION & EXPONENT BOTH IN SIGN & MAGNITUDE

I will assume an 8-bit fractional part Sign & Magnitude and a 4-bit exponent also represented in Sign & Magnitude.

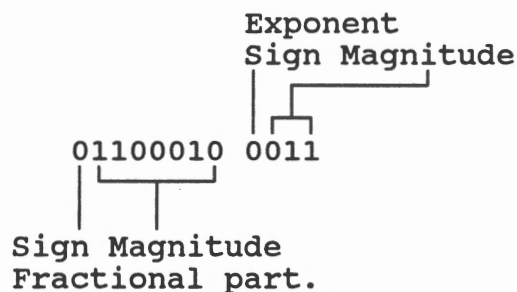
Example (a)

$$(6.125)_{10} = +(110.001)_2$$

$$\begin{aligned} \text{Normalize} \quad &= +(.110001 * 2^3)_2 \\ &= +(.110001 * 2^{+011})_2 \end{aligned}$$

$$\begin{aligned} \text{Express in} & \\ \text{Sign \& Magnitude} & \\ \text{Representation} &= (0.110001 * 2^{0011})_2 \end{aligned}$$

Final Representation



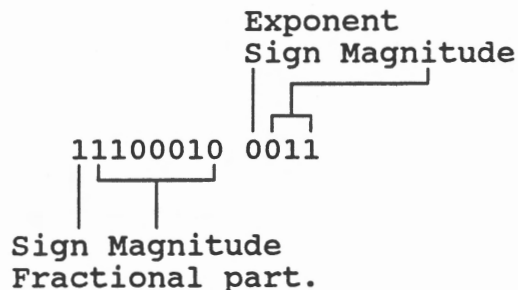
The binary point is implied as being immediately before the magnitude of the fractional part.

Example (b)

$$-(6.125)_{10} = -(110.001)_2$$

The only difference from example (a) is that the sign bit of the fractional part will be negative (ie 1).

Final Representation



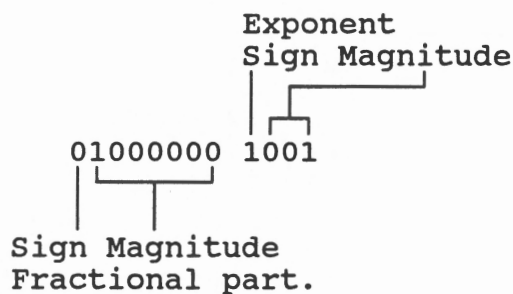
Example (c)

$$(0.25)_{10} = +(.01)_2$$

$$\begin{aligned} \text{Normalize} &= +(.1 * 2^{-1})_2 \\ &= +(.1 * 2^{-001})_2 \end{aligned}$$

$$\begin{aligned} \text{Express in} & \\ \text{Sign \& Magnitude} & \\ \text{Representation} &= (0.1000000 * 2^{1001})_2 \end{aligned}$$

Final Representation



The binary point is implied as being immediately before the magnitude of the fractional part.

Example (d)

$$-(0.25)_{10} = -(.01)_2$$

Only only difference to example (a) is that the sign bit of the fractional part will be negative (ie 1).

11000000 1001

the range of values we can represent with 24 bits (remember implied leading 1) and 7 bit exponent?

Range is given by $(R^{\text{no of bits}} - 1)$ ie $2^{24} - 1$

Smallest number: $-(2^{24} - 1) \times 2^{127}$ Largest: $(2^{24} - 1) \times 2^{127}$

Smallest possible fractional part is

$$\frac{1}{2} \times 2^{-128}$$

As there is always a 1 in the normalised fractional part. $0.1_2 = \frac{1}{2}_{10}$

FRACTION & EXPONENT BOTH IN 2'S COMPLEMENT

I will assume an 8-bit fractional part 2's Complement and a 4-bit exponent also represented in 2's Complement.

Example (a)

$$\begin{aligned}
 (6.125)_{10} &= +(110.001)_2 \\
 \text{Normalize} &= +(.110001 * 2^3)_2 \\
 &= +(.110001 * 2^{+011})_2 \\
 \text{Express in Sign \& Magnitude Representation} &= (0.110001 * 2^{0011})_2 \\
 \text{Final Representation} &
 \end{aligned}$$

	Exponent
	2's Complement
01100010	0011
Fractional part.	
2's Complement	

The binary point is implied as being immediately before the second bit

Example (b)

$$\begin{aligned}
 -(6.125)_{10} &= -(110.001)_2 \\
 \text{Normalize} &= -(.110001 * 2^3)_2 \\
 &= -(.110001 * 2^{+011})_2 \\
 \text{Express in Sign \& Magnitude Representation} & \\
 \text{Exponent: } +011 &= 0011 \\
 \text{Fraction: } -(.110001) &= -0.1100010 \\
 &\quad 2's \text{ complement } 1.0011110 \text{ (flip all bits \& add 1)}
 \end{aligned}$$

Final Representation

	Exponent
	2's Complement
10011110	0011
Fractional part.	
2's Complement	

Example (c)

$$(0.25)_{10} = +(.01)_2$$

$$\begin{aligned} \text{Normalize} \quad &= +(.1 * 2^{-1})_2 \\ &= +(.1 * 2^{-001})_2 \end{aligned}$$

Express in Sign & Magnitude Representation

Exponent: $-001 = 1111$ (flip all bits & add 1)Fraction: $=(.1) = 0.1000000$

$$(0.1000000 * 2^{1001})_2$$

Final Representation

	Exponent
	2's Complement
01000000	1111
Fractional part.	
2's Complement	

The binary point is implied as being immediately before the second bit.

Example (d)

$$-(0.25)_{10} = -(.01)_2$$

$$\begin{aligned} \text{Normalize} \quad &= -(.1 * 2^{-1})_2 \\ &= -(.1 * 2^{-001})_2 \end{aligned}$$

Express in Sign & Magnitude Representation

Exponent: $-001 = 1111$ (flip all bits & add 1)Fraction: $=(.1) = 1.1000000$ (2's comp of 01000000)

Final Representation

	Exponent
	2's Complement
11000000	1111
Fractional part.	
2's Complement	

The binary point is implied as being immediately before the second bit.

BIASED EXPONENTS

A system sometimes used to represent exponents is the biased method. Figure 4.6 illustrates the method for a 4-bit exponent.

Binary representation of the exponent	True exponent	Biased form
0000	-8	0
0001	-7	1
0010	-6	2
0011	-5	3
0100	-4	4
0101	-3	5
0110	-2	6
0111	-1	7
1000	0	8
1001	1	9
..
1101	6	14
1111	7	15

FIGURE 4.6

Bias consists of adding 2^{n-1} (where n =no of bits in the exponent field) to the true exponent.

There are 2 advantages to the biased form of the exponent. Firstly the most negative exponent is represented by zero. Secondly the exponents are monotonic in their binary form. To increase the exponent by one just add 1 to the exponent. (Similarly subtract 1 to decrease exponent by one). In both cases the binary exponent behaves like an unsigned binary number.

IF we have 8 bits in the exponent field, then to find the true exponent, you subtract 2^7 (~~128~~)¹²⁸ from the number.

Number	0	128	255
True exponent	-128	0	127

REPRESENTATIONAL ACCURACY

When a decimal number is converted to binary several cases can occur:

1 Binary representation is exact and the number of bits required is \leq the number of bits of the fractional part. In this case total accuracy is preserved.

Assume a 4-bit exponent & 4-bit mantissa both using the Sign & magnitude representation.

$$2.5 = (10.1)_2 = .101 * 2^2 = 0.101 * 2^{+010} = \begin{matrix} \text{Fr} & \text{Exp} \\ 01010010 \end{matrix}$$

2 Binary representation is exact but the number of bits required is $>$ the number of bits of the fractional part OR the binary representation of the number is infinite. In these cases the extra bits are lost.

In both cases the number should be **ROUNDED** rather than truncated to give better accuracy. For example:(4-bit fractional part)

$$\begin{aligned} 0.1101101 &= 0.111 \text{ (rounded up) [ie .0001101-->.001]} \\ 0.1100111 &= 0.110 \text{ (rounded down)} \end{aligned}$$

Example: [Binary representation is infinite]

$$(0.7)_{10} = (0.1 \ 0110 \ 0110 \ 0110 \ \text{etc})_2$$

Fractional part = 0.1011 (.0000 0 0110 0110 etc
is truncated)

SOME TYPICAL FLOATING POINT SYSTEMS

The computer designer decides on a floating point representation for a specific computer and must decide:

1. The TOTAL number of bits to be used.
2. The representation of the exponent.
3. The representation of the fractional part (including normalization)
4. The number of bits for the exponent & for the fractional part.
5. The location of the exponent (exponent first or fractional part first).

IEEE FLOATING POINT FORMAT

The Institute of Electronic & Electrical Engineers has produced a standard floating point format for mini & micro computers.

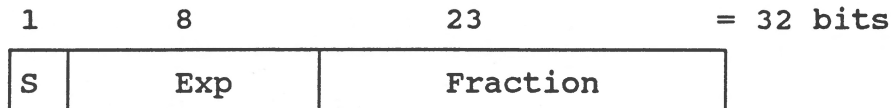
$$X = (-1)^S * 2^{(E-127)} * (1+F)$$

The fraction ($f=1+F$) is in the range $1 \leq f < 2$

S=Sign bit 0= +fraction, 1= -fraction

E= 8-bit exponent biased by 127

F= 23-bit fraction(together with an IMPLICIT leading 1)



Example:

The representation of $-(2346.125)_{10}$ on a 16-bit wordlength machine.

$$\begin{aligned} -(2346.125)_{10} &= -(100100101010.001)_2 \\ &= -(1.00100101001010)_2 * 2^{11} \end{aligned}$$

The fractional part is negative thus $S=1$

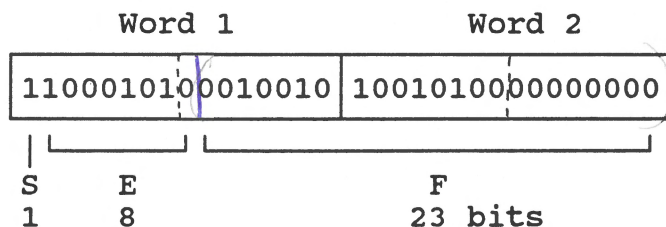
The fractional part is .00100101001010000000000 (in 23 bits)
Note that the leading 1 is omitted. It is IMPLICIT in this representation).

The exponent $E = 127 + 11 = 138 = (10001010)_2$

The final 32-bit representation is:

11000101000100101001010000000000

It is stored in 2 words each of 16-bits



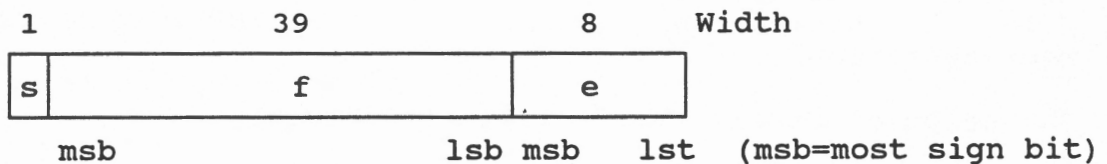
BASIC IEEE FLOATING POINT FORMATS

The 3 basic formats are given in Figure 4.7.

Type	Single	Double	Quad
Field width in bits			
S=Sign	1	1	1
E=Exponent	8	11	15
L=Leading bit	1	1	1
F=Fraction	23	52	111
Total Width	32	64	128
Sign bit	For all: 0= +, 1= -		
Exponent			
Maximum E	255	2047	32767
Minimum E	0	0	0
Bias	127	1023	16383

FIGURE 4.7**REALS IN TURBO 5**

A REAL number is represented in 48-bits. Otherwise the IEEE standard single format is used.



The value v of the number is determined by

IF $0 < e \leq 255$ THEN $v = (-1)^s * 2^{(e-127)} * (1.f)$.

IF $e = 0$ THEN $v=0$.

FLOATING POINT ARITHMETIC

Because real numbers are represented as a fractional part & an exponent arithmetic can not be done directly on the bits of the word. Any arithmetic operation takes several steps. The major aspects will be illustrated in a series of examples. For ease a decimal example will be used. The same considerations occur when the binary representation is used.

Example: Add $A = 94321.0$ and $B = 7240.0$.

If we were to add these numbers by hand we would:

$$\begin{array}{r} + 94321.0 \\ + 7240.0 \\ \hline +101561.0 \end{array}$$

We recognise that when the numbers are held in floating point form that for addition (& subtraction) the exponents of both numbers must be the same. Also we recognise that, in distinction from integer arithmetic, real arithmetic is a multistep process.

ALGORITHM FOR ADDITION & SUBTRACTION OF REAL NUMBERS

- 1 Identify the number with the smaller exponent.
- 2 Make this exponent the same as the exponent of the larger number. Adjust the fractional part accordingly.
- 3 Add (or subtract) the fractional parts.
- 4 If necessary normalise the result

Example: Add $A = .94321 * 10^5$ and $B = .724 * 10^4$

$$\begin{array}{rcl} A = .94321 * 10^5 & = & .94321 * 10^5 \quad (\text{Normalised}) \\ B = .724 * 10^4 & = & .0724 * 10^5 \quad (\text{Exponent same as above}) \\ & & \hline & & +1.01561 * 10^5 \\ & & = +.101561 * 10^6 \quad (\text{Post-normalisation}) \end{array}$$

The post-normalisation is necessary to retain as great a degree of accuracy as possible.

EXAMPLES

1 Complete the following table:

Decimal	Binary	Octal	Hexadecimal
3.6 .63	101.011 11011.01	4.3 .764	2.A5 AB.D

2 Do the following calculations for the values represented in (a) Sign & Magnitude, (b) 2's Complement & (c) 1's Complement. In all cases assume a 6-bit representation.

(1) $23 + 7$ (2) $-3 + 9$ (3) $25 - 17$ (4) $-13 - 11$

3 Given the following floating point representation:

S	Exponent	Mantissa
15 14	10 9	0

Where: S indicates the sign of the mantissa

The exponent is represented in 5 bits biased by 16.

The mantissa is represented in 10 bits, normalised with an explicit leading 1 after the point (0.1xxx)

(ie 2.0 --> 0100101000000000
 -2.0 --> 1100101000000000
 .5 --> 0100001000000000
 .125 --> 0011010000000000)

Give the floating point representation of the following decimal numbers in binary:

(a) 18.0625
 (b) 0.014
 (c) 0.501708984375

4 Start with the answer for 3c above, and calculate its decimal value. Explain why you did not get the original decimal number.

CHAPTER 5

HARDWARE IMPLEMENTATION

INTRODUCTION

The theme of this section is to describe how the various sub-systems of a computer are physically implemented. Cost and speed of various units is discussed. The important features, of the most common implementations of each sub-system, are described. Figure 5.1 is a reminder of the basic van Newman architecture.

BASIC VAN NEUMANN ARCHITECTURE

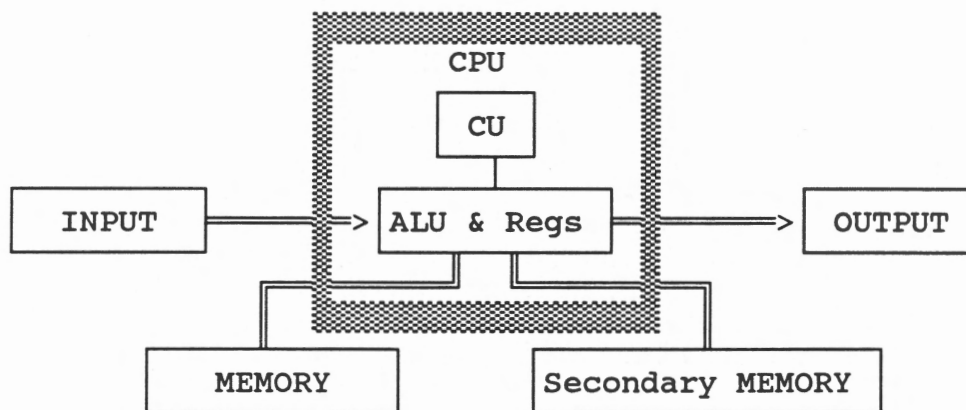


FIGURE 5.1

CENTRAL PROCESSING UNIT (CPU)

The CPU consists of: the Control Unit, registers and an arithmetic logic unit.

CONTROL UNIT

The Control Unit (CU) controls the operation of all the other units of the computer system. The CU causes the program instructions to be executed one after another. A program instruction is moved from memory to the instruction register in the CPU. Then the CU causes this instruction to be executed. The program counter is incremented and this entire cycle repeated endlessly. The CU also controls the action of all the peripheral units.

The CU consists of:

- (a) a complex electronic circuit that causes the actions described above to be performed by appropriate circuits; and
- (b) an electronic clock which produces square wave pulses. These are used to ensure that the signals given by the CU are applied to the appropriate circuit in the computer for long enough for that operation to be fully carried out. This clock rate is often quoted as the speed of the computer. Note that you can not just increase the speed of this clock infinitely because there is some critical minimum time required by an electrical circuit to perform its task.

REGISTERS

The CPU registers always have the fastest response time of any hardware in the computer ie the time taken to move the contents of a CPU register either to another CPU register or to the ALU is always the fastest operation that the computer can do. Each CPU register is made up of (often 8 or 16) individual bits. Each bit is made up of a flip-flop circuit. Each flip-flop circuit is made up of primary NAND, NOR, AND, OR or NOT boolean circuits. Each flip-flop can store the state of the bit, it can also pass this value on to some other flip-flop when required to do so.

ARITHMETIC LOGIC UNIT

This unit does all the arithmetic and logic functions of the computer. Today such a unit is usually composed of off-the-shelf Integrated Circuits (IC) which can perform all the required operations. In essence the IC receives input from 1 or 2 CPU registers and sends output to another CPU register. The IC itself is made up of AND, OR, NAND, NOR and NOT boolean circuits.

MEMORY

There are a number of memory devices available for storing information in a computer system. The main criteria for selecting a particular memory device over another is a trade-off between cost and performance. Performance is measured in terms of speed and capacity.

Main memory (MM) is the primary memory device in a computer system. Unlike secondary memory devices eg disks and tapes, which offer high capacity at a slow access speed for long term storage, main memory provides access times comparable to the working speed of the CPU and is thus the only memory device that communicates directly with the CPU. Main memory acts as a temporary intermediate device storage facility for programs and data during execution.

An earlier type of main memory known as core memory consists of a matrix of ferrite rings with several conductors threaded through each. The rings may be magnetized in either direction and each represent one bit. Core memory suffers the disadvantage that the process of reading the 'value' of a core destroys the state of the core. This necessitates an extra process to return the core to its original state. This naturally slows the process down. See Figure 5.2.

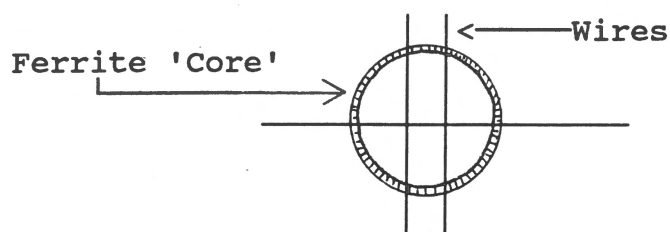


FIGURE 5.2

Core memory has now been replaced by semiconductor memory which consists of a large number of circuits (transistors) on a silicon chip, each circuit capable of being in an on or off state.

The cost of manufacturing semiconductor memory has decreased to well below that of the earlier core memory which is now no longer used, although its name lives on; main memory still widely being referred to as 'core', regardless of construction. Semiconductor memory has replaced core memory because it is cheaper to manufacture, has faster access times (ie. is quicker), has a non-destructive readout (ie. reading does not destroy the contents of memory), and is smaller (ie. permits greater bit densities). One advantage of core memory is that it is non-volatile (ie. does not lose its contents when power is removed). Semiconductor memory can be static or dynamic. The dynamic memory is volatile (ie the charge leaks away) and needs to be refreshed at regular intervals (every few micro-seconds).

CACHE MEMORY

Within the CPU of a computer there are also a fixed number of special very fast locations called registers. These are used for very fast temporary storage, typically by the ALU during expression evaluation. In a computer where only one operand is explicitly mentioned in an instruction (ie. a 1-address computer), a single special register known as the accumulator exists for holding intermediate results during a computation by the ALU. Sometimes main memory is too slow relative to the speed of the registers, and a small amount (typically 1-4KB) of very fast (and expensive) cache memory is used to act as a buffer between the CPU and main memory.

Typically 4KB of the program being executed is loaded from main memory to cache. The CPU registers then interface with the fast cache memory. At the end of the program the cache memory is copied back to main memory. More complex strategies for maximizing cache usage are used when the program is larger than the cache or when several programs are being executed by the CPU.

PERIPHERAL DEVICES

Input, output and long-term storage devices are collectively referred to as peripherals. They can be split into two 'types', namely offline and online devices. Offline devices read and write data from and to an intermediate storage medium (eg. magnetic tape, floppy disk) which is not all available to the computer system at the same time. Online devices (eg. video terminals) are directly accessible at all times. The most common long-term storage devices are magnetic tape and disk drives, which can be used for input and output. 'Long-term' refers to anything from a few minutes (eg. a temporary file), to several years (eg. an archive tape).

Each peripheral device has its own advantages or disadvantages relating to the way in which the data using them has to be organized. Thus the suitability of a peripheral depends on the particular application.

Two important terms associated with storage devices are 'sequential' and 'random'. Sequentially ordered data must be accessed from the beginning to the end of the data in the order that the data appears in the file (eg. magnetic tape). Randomly stored data may be accessed in any order (eg. disk drives).

PUNCHED CARDS AND PAPER TAPE

Both these media are no longer used widely. The most important disadvantages are:

- Correction of errors is time-consuming and wasteful
- Can easily be damaged by mis-use
- Access is only sequential
- Input and output speed is slow
- Cumbersome to handle
- Capacity is low relative to size of media
- Media is not reusable

MAGNETIC TAPE

Magnetic tape is a plastic tape covered with a magnetic surface. Data is recorded across the tape one character at a time. Each bit is recorded by magnetising an area of the tape in one of two directions. Typically tapes have 6 or 8 tracks with an extra track for a parity check. In Figure 5.3 below there are 8 data tracks with the data 10110100 and 1 parity track (with a 1bit). Commonly 1600 characters per inch are recorded.

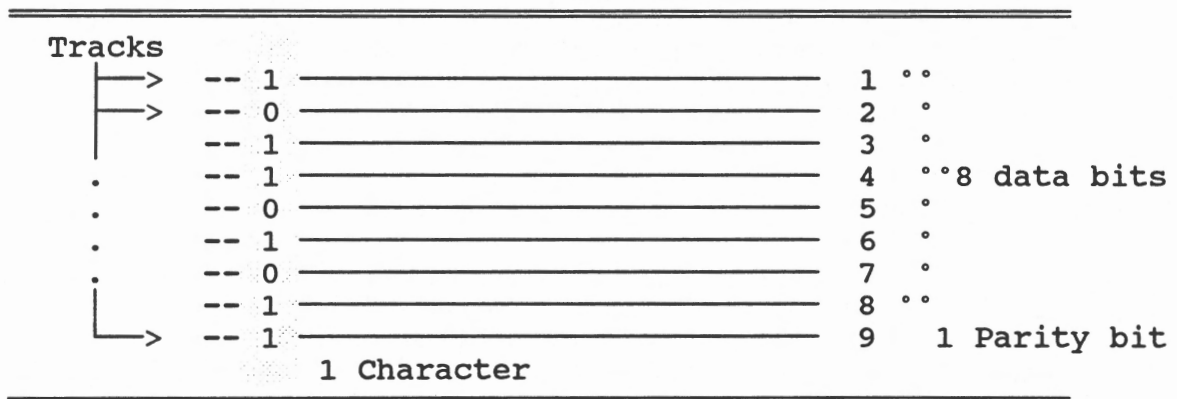


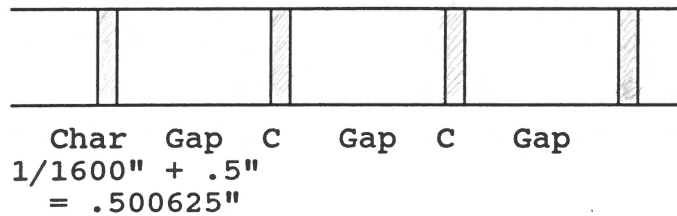
FIGURE 5.3 -- 9 TRACK MAGNETIC TAPE

TAPE DRIVES

A tape drive reads/writes data to/from the tape. Usually this is done at high speed. Because any drive of this type takes a certain time to get up to speed or to stop a GAP has to be inserted between the data that is read/written. This gap is usually .5" wide. Patently this gap is wide compared to the width of 1 character. Hence it pays to BLOCK many characters together.

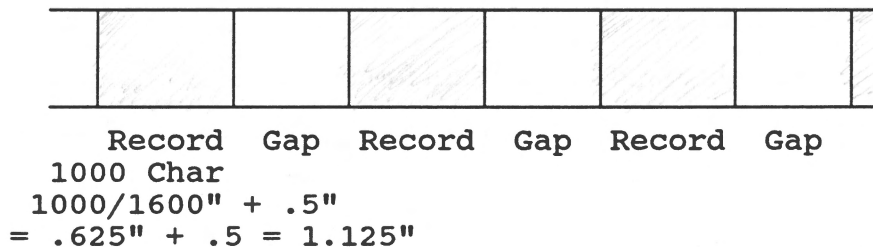
INTER-RECORD GAPS

The extreme case of recording 1 character at a time would lead to:



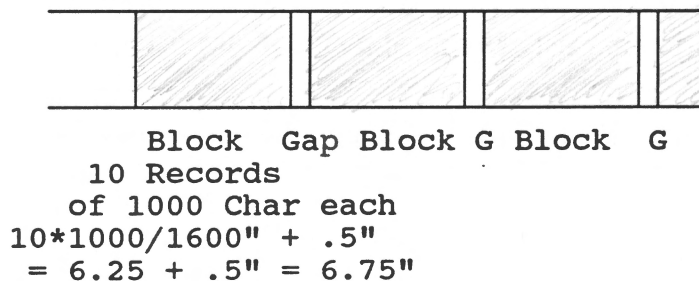
Patently the tape would be mainly $(.5/.500625 = 99.9\%)$ empty

Lets store 1 record, rather than 1 character. If the record has 1000 characters then we would have:



We are better off. 56% of the tape holds useful information but 34% is still empty.

Lets store 1 block of 10 records, rather than 1 record. If the record has 1000 characters then we would have:



We are well off now. 93% $(6.25/6.75)$ of the tape holds useful information only 7% is empty.

BUFFERS

Why do we not just make our block some very large number of records and consequently increase the tape useage to close on 100%? The reason is that the entire block has to be read from tape to memory at one time. A buffer area in memory, the same size as the block, must exist. If we make the buffer too big we can run out of memory space.

OTHER CHARACTERISTICS OF MAGNETIC TAPE

This medium is widely used in micro, mini and mainframe computer systems for off-line storage. It is a convenient medium for transportation over long distances , provides high capacity at low cost, and is portable (in that a number of different computer systems can read the same tape because the data is stored in a standard format).

- Used for input, output and storage
- Record size not limited
- Insertion, deletion, and alterations to records can only be performed by copying to another tape
- Access to information is only SEQUENTIAL
- A single tape holds a large volume of data (megabytes)
- Information is usually stored in a standard format
- Speed of access can go up to 300 000 chars/sec, which is much faster than punched cards and paper tape
- Magnetic tape is reusable

PRINTING DEVICES

Printers are the principal means of providing a 'hard copy', on paper, from a computer (Microfiche is also possible).

Many different types of printers exist. The most common are listed below together some comments on each.

Printers vary in price from R500, for a slow dot matrix printer, to R500 000 for an ultra fast printer.

Printers can be classified as impact or non-impact

DRUM or LINE PRINTER -- The printer paper passes over a print drum which rotates. The drum spins at high speed. The entire character set exists on the circumference of the drum for each of the print columns. When the appropriate character is in position a hammer strikes the paper leaving a carbon image of the character. Such printers commonly have 132 columns per line. Line printer speeds vary from 100 - 3000 lines/min.

DOT MATRIX PRINTER -- Specific needles from a rectangular matrix are magnetically energised to print the specific character required. Each character is printed sequentially from one side of the page to the other. A characteristic speed is 120 lines/min (2 pages/min).

INK-JET PRINTER -- It is similar to a CRT in operation. A fine jet of ink is emitted from a tiny nozzle. This creates a high-speed stream of ink drops. The nozzle is vibrated ultrasonically to ensure that the ink stream is broken up into individual drops. Each drop is given an electric charge as it leaves the nozzle. The stream of drops is then deflected electrostatically by vertical & horizontal deflection plates (just like a CRT). By moving the beam, characters are written on to the surface of the paper. By arranging the paper to be off the central axis of the beam the ink drops that are not deflected at all do not strike the paper & are collected in a reservoir for reuse.

LASER PRINTER -- This is expensive but very versatile. Exceptionally high quality printing is possible. A large number of different fonts are available.

The laser printer prints using a high-resolution dot-matrix pattern (300 x 300 dots per inch is characteristic). The principal of operation is: A low-power laser is used in conjunction with an electrical corona to impart an electrical charge to the surface of a revolving photosensitive 'print-drum'. The toner, a dry powdered substance is attracted to this charge, which is in exact representation of the printed image. The toner is then transferred to the surface of the paper and fused (melted) into place. A representative speed is 8 pages per minute (500 lines per minute).

VDUS AND TERMINALS

- A VDU (Visual Display Terminal) is a Cathode Ray Tube & is the most common terminal in present day usage.
- It is a convenient input/output device for users.
- It is an online device, therefore fast data retrieval and easy editing are characteristic.

DISKS (AND DISK DRIVES)

A Disk is a flat circular surface which is coated with a magnetic material. The disk drive spins this disk at high speed. On each disk surface there are many concentric tracks on which the data is stored. The read/write head is first positioned on the correct track and then the data required is read onto/off that track. See Figure 5.4.

Both **FIXED** and **REMOVABLE** disks exist as do **SINGLE** and **MULTIPLE** platter disks. For multiple platter disks there is a R/W head for each disk surface. See Figure 5.5. Note that each track contains the **SAME** amount of information. The **DENSITY** of storage is higher on the smaller tracks

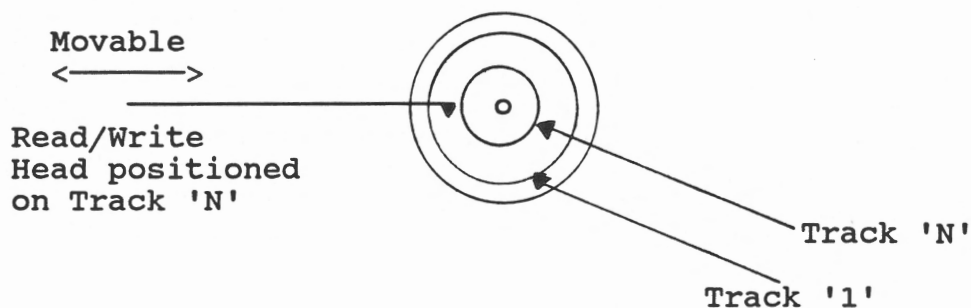


FIGURE 5.4 -- SINGLE PLATTER DISK

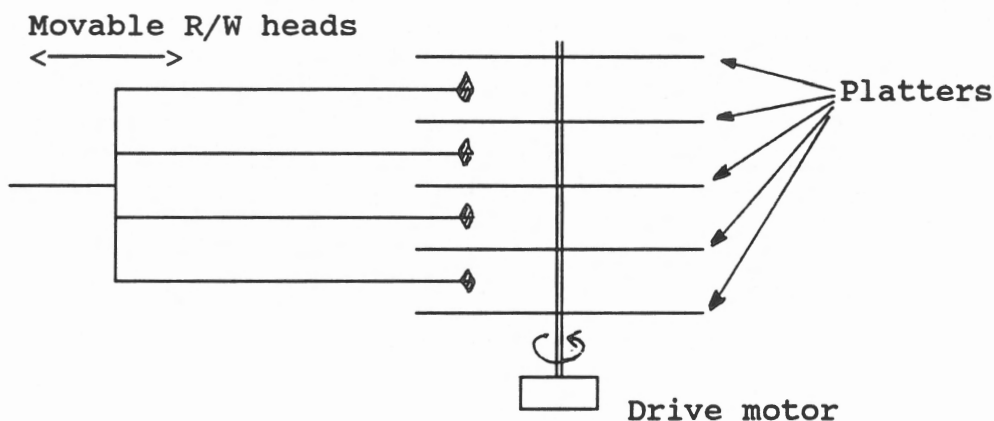
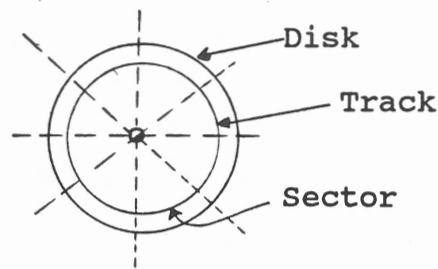


FIGURE 5.5 -- MULTIPLE PLATTER DISK

A **CYLINDER** is defined as 1 track on each platter ie Cylinder 3 is track3 on each and every platter. It is possible to write/retrieve all the data on a cylinder without any movement of the R/W heads. For optimised data transfer all the data accessed should ideally be on one cylinder. The Top surface of the top platter and the bottom platter of the bottom surface are not used due to potential damage to these surfaces.

SECTORS



- EACH TRACK is subdivided into some integral number of SECTORS of the same size.
- The SIZE of the sector is usually predefined by the manufacturer (called HARD sectoring). On some systems the programmer is allowed to subdivide the track (SOFT sectoring).
- Information is moved to/from the disk in UNITS OF 1 SECTOR. A BUFFER store, the same size as a sector, must exist in main memory to hold all this data.

TIME TO ACCESS DATA ON A DISK

To access data from a disk:

- 1 The R/W head must move to the correct track. This is called the SEEK TIME S_i .
- 2 The time for the disk to rotate so that the appropriate information is under the R/W head is known as the ROTATIONAL DELAY or LATENCY L_i . The AVERAGE LATENCY is the time it takes for the disk to do half a rotation.
- 3 The time taken to transmit all the data to/from the disk is called the TRANSMISSION TIME T_i .

$$T_i = \text{Amount of data to be transfered} / \text{Transmission Rate}$$

EXAMPLES

A disk has a SEEK time = 40msecs
 a ROTATIONAL speed = 50 revolutions/sec; and
 a TRANSMISSION speed = 180ch/msec.

Deductions:

1 revolution takes $1/50$ secs = .02 secs = 20 msecs

Thus it can store $20 * 180$ char/track = 3600 char/track

Average LATENCY = Time for half a revolution = 10 msecs.

Question 1:

How long does it take to access 3000 chars all on 1 track?

$$\begin{aligned}\text{Access Time} &= S_1 + L_1 + T_1 \\ &= 40 + 10 + 3000/180 = 50 + 16.7 = 66.7 \text{ msec}\end{aligned}$$

Question 2:

How long does it take to access 4500 chars?

First we assume a single platter disk.
1 track holds a maximum of 3600 chars thus we assume that the 4500 characters are kept on 2 different tracks.

$$\begin{aligned}\text{Access Time} &= S_1 + L_1 + T_1 + S_2 + L_2 + T_2 \\ &= 40 + 10 + 3600/180 + 40 + 10 + 900/180 \\ &= 50 + 20 + 50 + 5 = 125 \text{ msec}\end{aligned}$$

COMMON DISKS

There are 3 very common types of disks in usage today: HARD, FLOPPY and STIFFY. They all use the basic principles discussed above.

HARD DISKS

Hard disks exist both as fixed and removable disks. The removable disks can be single or multiple platter disks. Generally hard disks are the most expensive type of disk but have the largest storage capacity and are fastest. The R/W heads fly above the surface of the disk (never touching it)

FLOPPY DISKS

Floppy disks are made out of flexible plastic and come enclosed in a paper envelope with a window to allow read/write operations. The R/W head actually makes contact with the disk surface thus these disks have a limited lifespan (± 100 hours). Floppy disks are removable.

STIFFY DISKS

Stiffy disks have characteristics between the above two types. The R/W heads fly above the disk surface. The disk is small (± 3 " diameter) and rigid. It has a solid hub at the centre (in contrast to the floppy disk) and rotates faster. It has a larger capacity and is removable.

SOME RELATIVE DISK CHARACTERISTICS

Naturally speeds & costs vary with manufacturer and economic climate. The table below is intended to give the reader an overview only. Single density floppy & stiffy figures are given. Double & quad density does exist. The magnetic surface of such disks has to be of better quality.

	HARD	FLOPPY 5 1/4"	STIFFY
CHARACTERISTIC			
Cost	--	R2	R7
Disk Drive Cost	>R1200	R600	R900
Rotational Speed	±60revs/sec	6 revs/sec	12 revs/sec
Seek Time	20 msec	170 msec	100 msec
Latency	8 msec	83 msec	40 msec
Capacity	5-200M Chars	360K Chars (Single density)	±700K Char

SOME ATTRIBUTES OF DISKS

- Direct access device (ie. RANDOM access to any piece of data is possible)
- Easy insertion of data possible
- Each removable disk pack can hold a large volume of data, making it ideal for long-term storage
- Used for input, output and storage
- Very fast relative to other secondary storage media
- Can easily be damaged by mishandling

EXAMPLES

1 A magnetic tape has a packing density of 800 characters per inch, an interrecord gap of $1/2$ ", and is filled with records. Each record contains 400 characters. Calculate the fraction of the tape containing useful data if the records are written as (a) Single record blocks; and (b) Blocks containing 4 records.

2 A single surface disk has an average seek time of 60 milliseconds (ms), a rotational speed of 2400 revolutions per minute, and can store 65536 bits per track. What is the average access time in milliseconds to read 10Kbytes of data. You may assume the data is written on as few tracks as possible. Note $1K = 1024$.

CHAPTER 6

THE COMPUTER & THE ASSEMBLER

INTRODUCTION

In this section the fundamental structure of a computer will be explained and discussed. Interwoven will be a discussion of Binary Machine Code (BMC) and of Assembler code. This interleaved structure has been chosen purposely because once the basic architectural principle has been introduced there is a need to communicate with that architectural feature. The appropriate Assembler instructions are then used in relevant examples to reinforce the principles discussed.

A simple computer, with its associated Binary Machine Code and Assembler, has been simulated on a standard PC. The entire system is called SMALL and is used in these notes. I draw the readers attention to the fact that if they were using a real computer they would have to press buttons etc on the real computer. In SMALL all these functions exist and are chosen as options from a menu.

STRUCTURE OF A SIMPLE ELECTRONIC COMPUTER

The basic units of an electronic computer are:

- Console
- Input device
- Output device
- Memory
- Central Processing Unit (CPU). Comprising:
 - Arithmetic & Logic Unit
 - Control Unit
 - CPU Registers

The inter-connection of these units is shown in Figure 6.1.

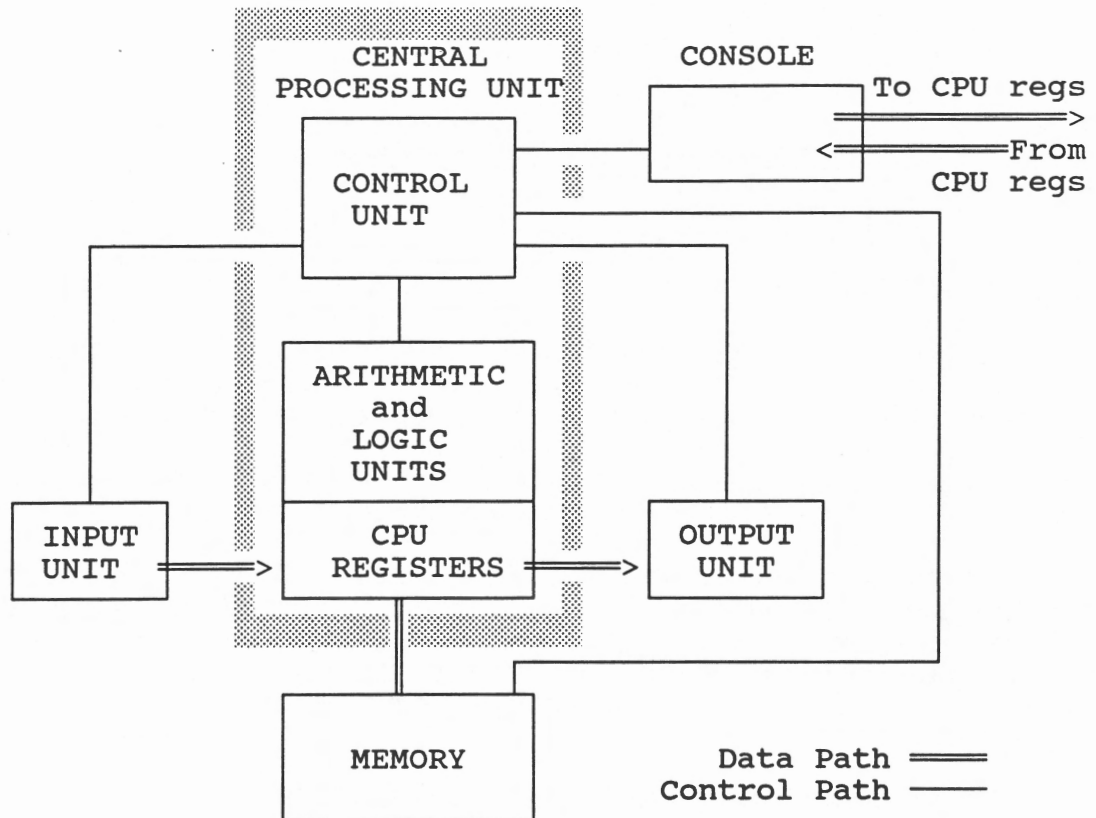


FIGURE 6.1

MAIN MEMORY

The main memory of a computer is the unit that stores both the **PROGRAM** that is to be executed as well as the **DATA** pertaining to the problem. The main memory is made up of individual words. A word is usually some multiple of 8 bits. Each word can hold either a computer instruction or some data value. The contents of a word of memory can be sent to some register in the CPU or vice-versa. The main memory of **SMALL** consists of 256 (1/4K) words each of 16 binary bits.

DATA

2's complement representation is used.

For negative values
For positive values

$$\begin{aligned} -K &= 2^{16/2} - K = 2^8 - K \\ K &= K \end{aligned}$$

For example:

5 is represented as: 0000000000000101

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1

-5 is represented as: 1111111111111011

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1

Sign bit

Effectively bit 15 is the sign bit.

INSTRUCTIONS

SMALL has a 1-address instruction format (ie an instruction contains only one main memory address).

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	Op code				II		MM		Address							

The Op Code field has 4 bits. This allows $2^4 = 16$ unique Op. codes. (ie Op codes 0 to 15 0000 to 1111)

The Address field has 8 bits allowing $2^8 = 256$ address to be specified. (ie addresses 0 to 255 0000 11111111)

The use of bits 8-9 and 10-11 will be explained later.

CONSOLE

The console of a computer contains a variety of switches and registers that allows the user to interact with the computer. The most important of these are:

On/Off switch. Used to turn the computer on/off.
Stop/Start switch. Used to start/stop the computer running.
Single step/Run to completion switch.

In the single step mode only one instruction is executed. This is very useful when debugging a program. In the run-to-completion mode the computer executes instructions sequentially until a HALT instruction is executed.

16 bit DISPLAY register.

This register can be used to display the contents of some word of memory or CPU register. Values can also be inserted into this register for subsequent transmission into the computer. For each bit of this display register there is a switch that can be used to set the value of the specific bit of the register.

Mode selector switch.

The following different modes may be chosen:

- (a) Enter data into the display register from the switches.
- (b) Enter contents of display register to the Memory Address Register.
- (c) Enter the contents of the display register to a specified word of memory.
- (d) Load the contents of a specified word of memory to the display register.
- (e) Use the display register to specify a specific CPU register.
- (f) Load the contents of a specific CPU register to the display register

SMALL is executed on a simulated computer (ie there is a program inside the computer that acts as (simulates) another computer). Consequently the console of the real computer is not available to the user. All the functions of the console, described here, are also simulated and available to the user when a SMALL program is running (to be run). Note that many assemblers today exist in this simulated fashion. This is because it is much easier for the user to use the simulator. Also far better error detection and correction features are usually implemented. This holds true both for the assembler and for the hardware.

CENTRAL PROCESSING UNIT

The CPU consists of several registers and the Arithmetic & Logical Unit.

CPU REGISTERS

The registers of the CPU are shown in Figure 6.2. (All the registers of SMALL are 16 bits unless stated otherwise). Also shown are the important interconnections between these registers and the rest of the computer.

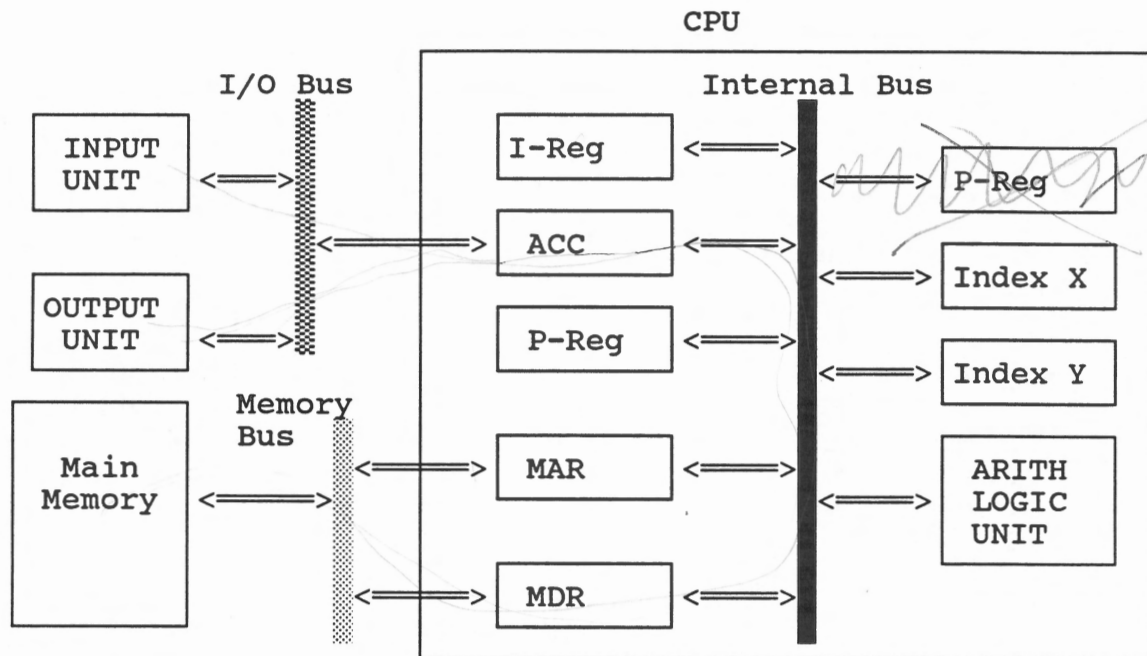


FIGURE 6.2

MEMORY ADDRESS REGISTER (MAR): The contents of this register specifies which word in memory is indicated.

MEMORY DATA REGISTER (MDR): This register is used to store the value being transferred to/from the specified word of memory.

INSTRUCTION REGISTER: This register holds a copy of the instruction currently being obeyed. *8 bit?*

PROGRAM REGISTER (LOCATION COUNTER): This register holds the Main Memory address of the instruction currently being obeyed.

ACCUMULATOR: This register is used to store any value to be manipulated. This register is used to input a value to the ALU. It is also used to receive the output from the ALU.

INDEX REGISTERS: The contents of these registers can be used to Modify the address of an instruction (more about their usage later).

THE BASIC INSTRUCTIONS FOR SMALL

The EXECUTABLE instructions of SMALL are given in Figure 6.3. The NON-EXECUTABLE instructions are given in Figure 6.4.

INSTRUCTION	OP CODE	NO	BINARY
Halt	HALT	0	0000
Increment Accumulator	INC	1	0001
Increment X Index Reg	INC @X		000101
Increment Y Index Reg	INC @Y		000110
Add	ADD	2	0010
Store	STORE	3	0011
Load	LOAD	4	0100
Jump	JMP	5	0101
Jump Equal	JEQ	6	0110
Jump Greater Than	JGT	7	0111
Jump Less Than	JLT	8	1000
Subtract	SUB	9	1001
Input	INP	10	1010
Output	OUTP	11	1011
Shift/Rotate		12	1100
Shift Left	SHL		110000
Shift Right	SHR		110001
Rotate Left	ROL		110010
Rotate Right	ROR		110011
Boolean		13	1101
AND	AND		110100
OR	OR		110101
XOR	XOR		110110
NOT	NOT		110111
Call a procedure	CALL	14	1110
Return from procedure	RET	15	1111

FIGURE 6.3 -- EXECUTABLE INSTRUCTIONS

INSTRUCTION	OP CODE
Block Storage Start	BSS
Define a value	DEC
Define a string	STR
Define a constant	CONST
Define an Address	ADDR
End of Assembler Prog.	END

FIGURE 6.4 -- NON-EXECUTABLE INSTRUCTIONS

These instructions can be divided up into different categories depending on what they do. These categories are given below, in Figure 6.5. The action of each instruction is described in greater detail.

DATA MOVEMENT

Memory -->Register	LOAD p	Put the value of Main Memory store 'p' in the ACC.
Register-->Memory	STORE q	Put the value in the ACC in store 'q' of Main Memory.
Memory -->Memory	None exists in SMALL	<i>we only have single address commands - we cannot do [A], [B]</i>
Register-->Register	LOAD Acc from index reg STORE Acc into index reg	

ARITHMETIC

ADD r	ACC := ACC + contents of the MM address 'r'
SUB r	ACC := ACC - contents of the MM address 'r'
INC	ACC := ACC + 1
INC @X	@X := @X + 1 (Index Reg X incremented by 1)

INPUT/OUTPUT

INP	Input an integer value to the ACC
INCH	Input a character to the ACC
OUTP	Contents of the ACC(an integer) is output
OUTCH	Contents of the ACC(a character) is output

CONTROL TRANSFER

JMP a	Jump to instruction labelled 'a'
JEQ a	If ACC=0 Jump to instruction labelled 'a'
	If ACC<>0 Jump to next instruction
JLT	If ACC<0 Jump to instruction labelled 'a'
	If ACC#<0 Jump to next instruction
JGT	If ACC>0 Jump to instruction labelled 'a'
	If ACC>=0 Jump to next instruction

PROCEDURE TRANSFER

CALL s	Jump to the start of procedure 's'
RET [s]	Return from procedure 's'

PROCEDURE DEFINITION

s PROC	Defines the start of procedure 's'
--------	------------------------------------

FIGURE 6.5 (CONTINUED OVER PAGE)

<u>DATA DEFINITION</u>		
BSS	Block Storage Start c BSS 4	Sort of array function. See Pg 66b Defines 4 stores. The name of the 1st store is 'c'
DEC	Define an integer ten DEF 10	Defines one store called 'ten' & initialises it to 10
STR	Define a string head STR 'CSC105W'	Defines a string called 'head' set to the characters CSC105W
CONST	Defines a constant max CONST 100	Defines a value that can not be changed
ADDR	Stores an address a ADDR b	Defines one store called 'a' & puts the address of 'b' into it
<u>PSEUDO OP CODES</u>		
END	End of the ASSEMBLER program. (This statement is needed by the assembler prog to determine the end of a user program)	

FIGURE 6.5

COMMUNICATION WITH MAIN MEMORY

The Memory Address Register (MAR) & the Memory Data Register (MDR) are used for communication between the CPU & Main Memory (MM). The Memory bus is used to pass the data between these registers and MM.

Example:

	MAR	MDR	Value in Word of MM
Store the value $(7)_{10}$ in word 2	2	7	? Before 7 After
Store the value $(9)_{10}$ in word 2	2	9	7 Before 9 After
Store the value $(6)_{10}$ in word 5	5	6	? Before 6 After
Load (Retrieve) the value in word 2 of Main memory	2	? Before 9 After	9

ARITHMETIC AND LOGICAL UNIT

This unit carries out the arithmetic and logical functions that the computer is capable of performing.

In overview the arithmetic functions that can be performed are only +, - in SMALL (On more general machines *, / will also be available). In the case of addition the contents of 2 registers of the CPU (The ACC and the MDR) are input to the ADD circuit which produces as output the sum of these values. This output is sent to some specific register of the CPU (usually the ACC).

A similar situation exists for the LOGICAL functions like AND, OR, NOR etc.

CONTROL UNIT (CU)

This unit controls exactly what the computer is doing at any specific moment in time. The CU has 3 flip-flops that determine its status. They are:

FLIP-FLOP	STATUS
RUN	ON/OFF
STATE	<div style="display: inline-block; border: 1px solid black; padding: 5px; text-align: center;"> <div style="display: flex; align-items: center; justify-content: center;"> <div style="text-align: right; margin-right: 5px;">FETCH EXECUTE</div> <div style="text-align: center;"> an instruction an instruction </div> <div style="text-align: left; margin-left: 5px;"> < > </div> </div> </div>
I/O TRANSFER	ON/OFF

When the RUN flip-flop is OFF the computer is idle. When this is turned ON then the Control Unit causes an instruction to be loaded from memory to the Current Instruction Register in the CPU. Then the Control Unit causes that instruction to be executed. This FETCH-EXECUTE cycle is continued until a HALT instruction is executed by the program (or the computer is switched off). The Control Unit also controls the action of the Input & Output Units. When an I/O instruction is executed the I/O flip-flop is turned ON. The CPU is then idle until the specific I/O device has finished its job and turned this flip-flop to OFF. It must be pointed out that an I/O operation usually takes a very large number of CPU cycles to complete.

There are two types of Control Units. They are known as SYNCHRONOUS and NON-SYNCHRONOUS.

SYNCHRONOUS (OR CLOCKED).

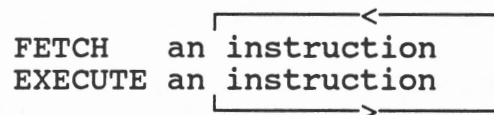
A basic fixed time period called a CYCLE is chosen. A cycle is long enough for any operation (excluding I/O ops) to be completed. Each cycle itself is broken up into several MINOR CYCLES. Each sub-event will start at the beginning of a minor cycle & the duration of the minor cycle is long enough for this sub-operation to finish. Should there be 8 minor cycles to a cycle and the specific operation only requires 5 to complete the last 3 minor cycles are just wasted (ie the computer does nothing during this time). SMALL has a Synchronous Control Unit.

NON-SYNCHRONOUS.

Each operation in the computer signals its successor when it is finished. Thus each operation only takes as long as is needed. There being no wasted time is the advantage of this method. The disadvantage is that each operation has to be intelligent enough to know when it has completed its task and signals the next one to start. The hardware to achieve this is complex, expensive and infrequently used.

BASIC OPERATION OF A SIMPLE COMPUTER

Assume that the RUN flip flop is switched to ON. The computer then alternates between the states



until a HALT instruction is executed (this turns the Run flip-flop to OFF). The detailed steps of the FETCH and EXECUTE cycles are given below.

FETCH CYCLE

The I-reg is loaded with the next instruction to be executed. The address of this 'next' instruction is specified by the P-reg.

INDIVIDUAL STEPS:

1. P-reg --> MAR (Mem Address Reg)
2. M[MAR]--> MDR (The contents of the word specified by the MAR is loaded to the MDR)
3. MDR --> I-reg
4. STATE --> EXECUTE

EXECUTE CYCLE

The instruction in the I-reg is executed.

INDIVIDUAL STEPS:

1. Instruction in I-reg is executed.
2. P-reg --> P-reg + 1
(Except for JUMP instructions when P-reg <-- Jump address)
3. STATE --> FETCH

A SIMPLE EXAMPLE

The following program reads in 2 values, adds them and output the sum. The Binary Machine Code as well as the Assembler code for this example is shown in Figure 6.6.

Binary Machine Code			Assembler
Op Code	Operand Address		
1010 0000	00000000		INP ;1st value
0011 0000	00000110		STORE X
1010 0000	00000000		INP ;2nd value
0010 0000	00000110		ADD X
1011 0000	00000000		OUTP
0000 0000	00000000	HALT	HALT — end of program
0000 0000	00000000		X BSS 1
			END — end of data area

FIGURE 6.6(A)

FIGURE 6.6(B)

Let us start at the beginning. We have a computer and **absolutely** no software. To write a program for this problem we first write the program in Binary Machine Code (as shown in Figure 6.6(A)). Then by using the display register on the control unit of the computer we load this program, instruction by instruction, into successive words of the computer memory. We then put the address of the first instruction of the program into the P-reg. of the computer and RUN the program.

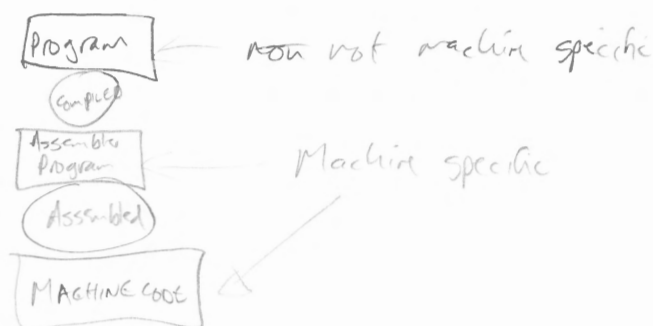
There are two major disadvantages. One: is the difficulty of writing in BMC. Its tricky and its easy to make mistakes. Also its darn difficult to interpret Binary code. Two: is the fact that we have to painstakingly, instruction by instruction, load this program to memory ourselves.

For the very first computers produced this is what had to be done. However this situation did not last for long.

Firstly: to ease the problem of hand 'loading' the program to memory a special "loader" program was written to do this automatically. This "loader" is written in BMC, is very short, and is 'hand' loaded to memory. It then reads in the user program, from some input device, and automatically stores the program in successive words of memory.

Secondly: An ASSEMBLER program was written (in BMC). This program allowed the user to write their program in assembler language, Figure 6.6(B). This notation is much easier to use. The ASSEMBLER program has to translate the assembler language into BMC so that the computer can understand & execute the instructions. (Details of structure of this ASSEMBLER program is given later in the text).

For ease of expression I shall write all programs in the "ASSEMBLER LANGUAGE" from here on because it is easier to understand.



The 'assembler' program has two more instructions than the BMC program. The **x BSS 1** instruction defines 1 store with the name 'x'. It does not exist in the BMC program because here the programmer, after finishing the program knew that the first empty store was store 6. The programmer then used store 6 for the data and directly entered the address 6 (0110) in the address part of the appropriate instructions. The **END** instruction is needed for the 'assembler' program to know that the users program is finished.

EXECUTION OF A PROGRAM

The first 4 instructions of the program given in Figure 6.6 (B) will be described in detail. The first instruction is **INP**.

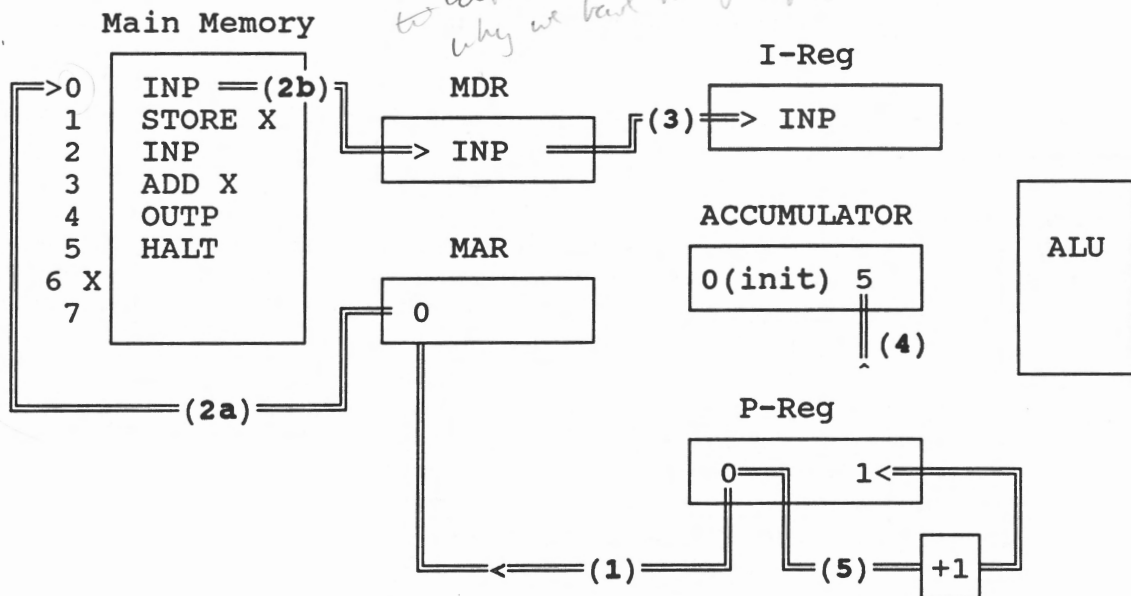
Initially $P = 0$ & $ACC = 0$

FETCH (1st INSTRUCTION)

1. $P \rightarrow MAR$ (1)
2. $M[MAR] \rightarrow MDR$ (2a) & 2(b)
3. $MDR \rightarrow I\text{-Reg}$ (3)
4. $STATE \rightarrow EXECUTE$

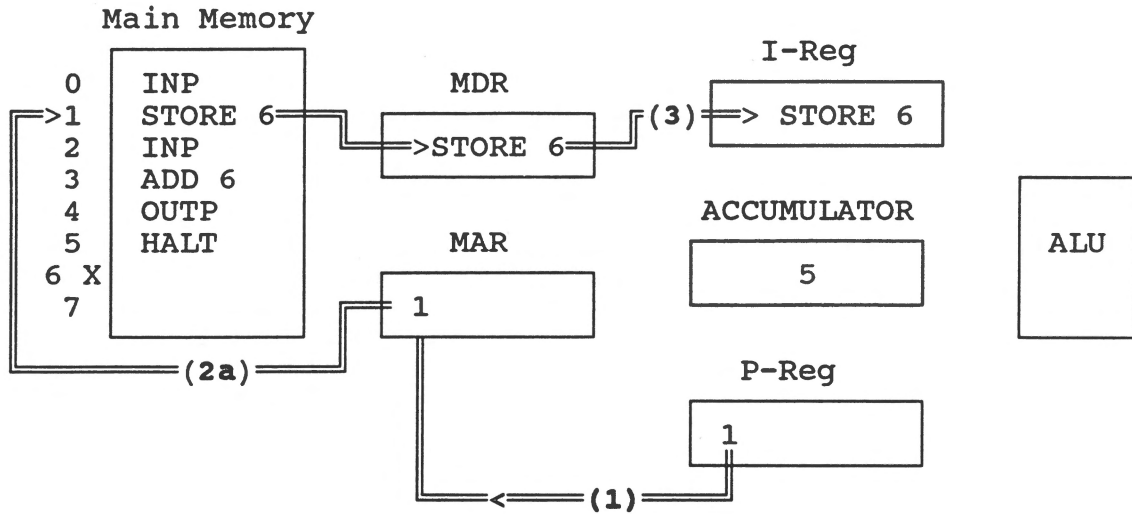
EXECUTE (1st INSTRUCTION)

1. Instruction in I-reg is executed (4)
(It is an INP instruction and the value 5 was read into the ACC.)
2. $P\text{-reg} \rightarrow P\text{-reg} + 1$ (5)
3. $STATE \rightarrow FETCH$



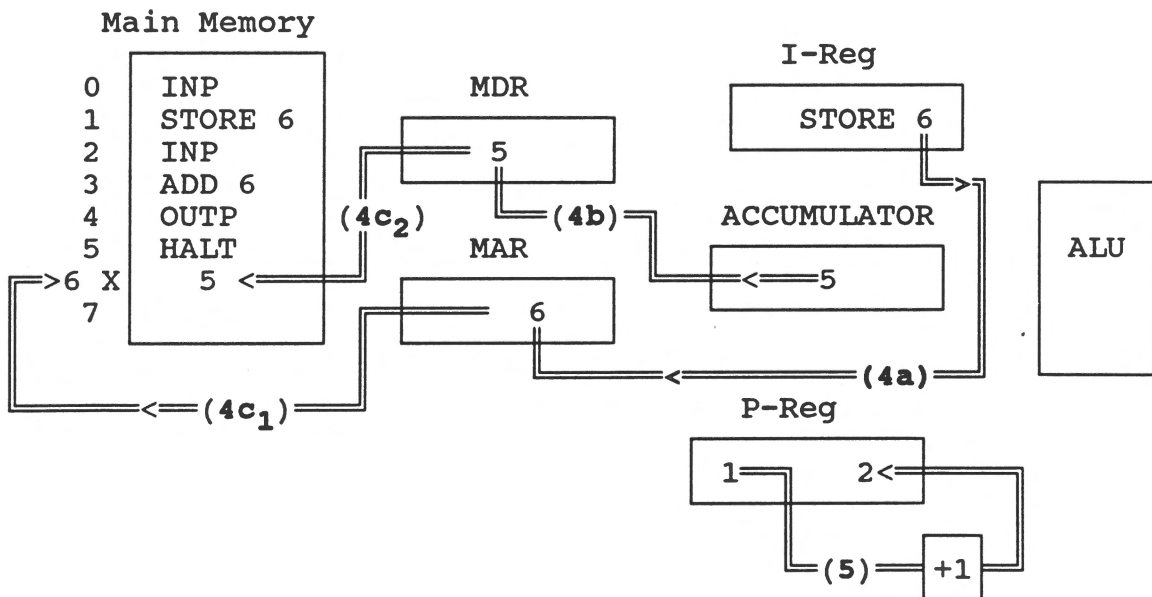
FETCH (2nd INSTRUCTION) ie The instruction STORE 6

1. P --> MAR (1)
2. M[MAR] --> MDR (2a) & 2(b)
3. MDR --> I-Reg (3)
4. STATE --> EXECUTE



EXECUTE (2nd INSTRUCTION) ie Execute instruction STORE 6

1. Instruction in I-reg is executed
(It is a STORE 6 instruction)
The address 6 --> MAR (4a)
ACC --> MDR (4b)
MDR --> M[MAR] (4c₁) & (4c₂)
2. P-reg --> P-reg + 1 (5)
3. STATE --> FETCH

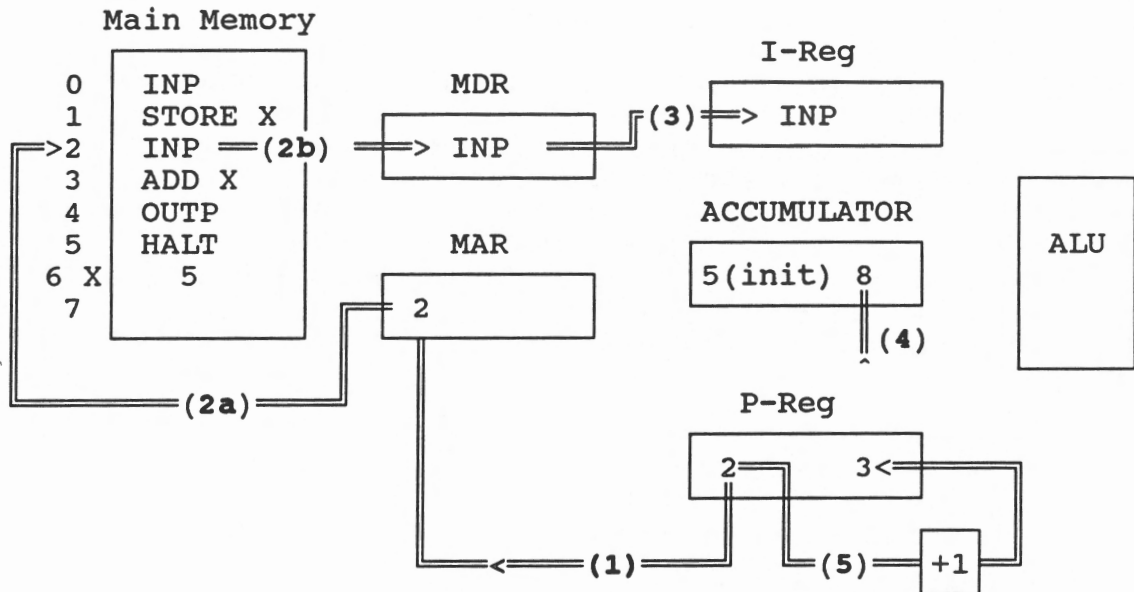


FETCH (3rd INSTRUCTION) Another INP instruction

1. P --> MAR (1)
2. M[MAR] --> MDR (2a) & 2(b)
3. MDR --> I-Reg (3)
4. STATE --> EXECUTE

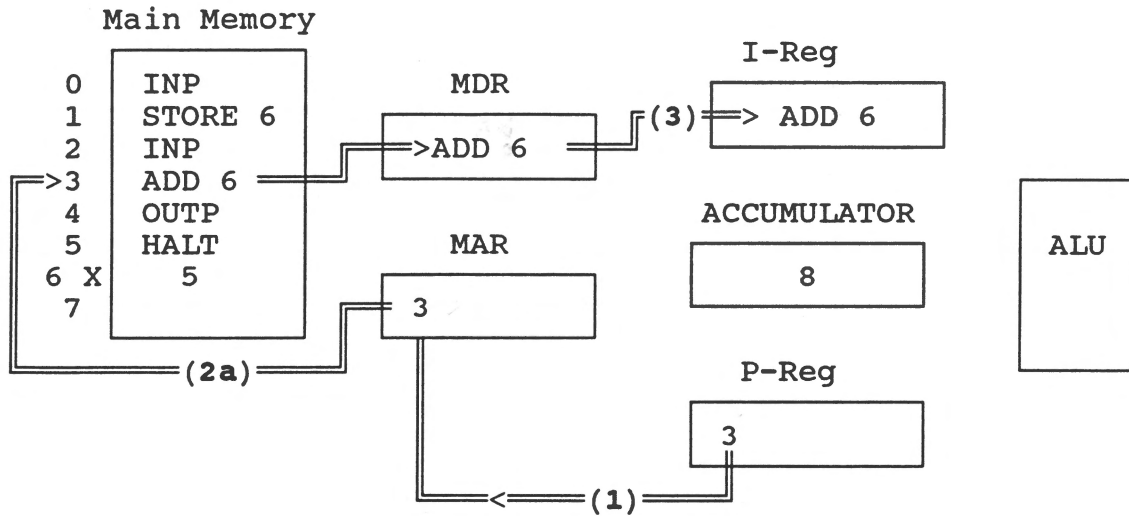
EXECUTE (3rd INSTRUCTION)

1. Instruction in I-reg is executed (4)
(It is an INP instruction and the value 8 was read into the ACC.)
2. P-reg --> P-reg + 1 (5)
3. STATE --> FETCH



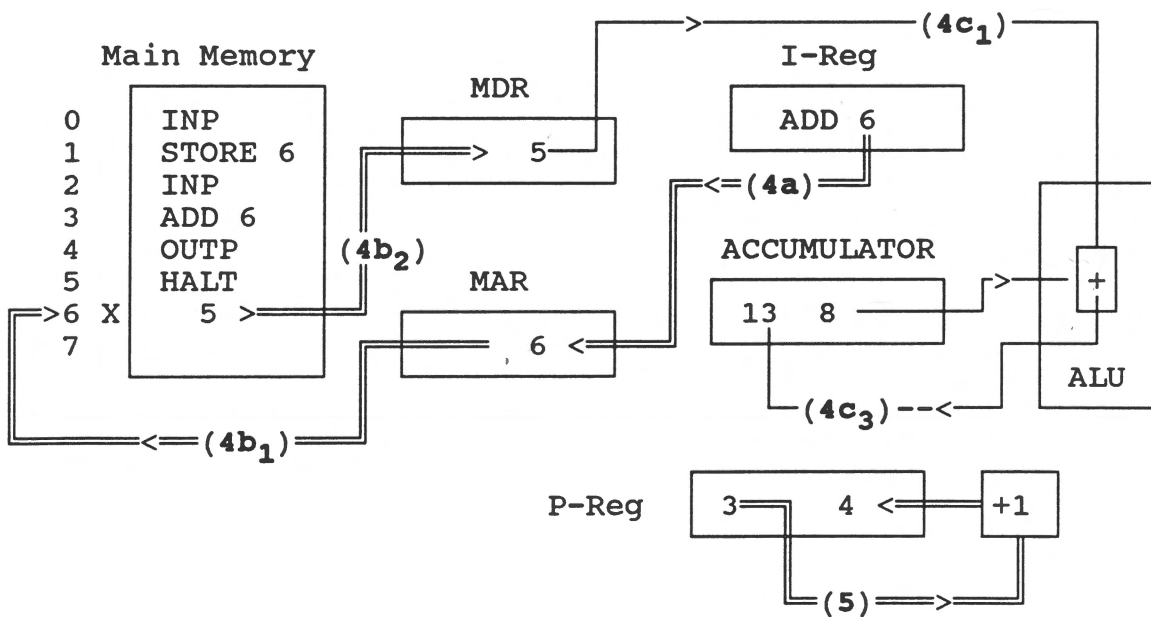
FETCH (4th INSTRUCTION) The instruction ADD 6

1. P --> MAR (1)
2. M[MAR] --> MDR (2a) & 2(b)
3. MDR --> I-Reg (3)
4. STATE --> EXECUTE



EXECUTE (4th INSTRUCTION)

1. Instruction in I-reg is executed
(It is a ADD 6 instruction)
The address 6 --> MAR (4a)
M[MAR] --> MDR (4b₁) & (4b₂)
ACC <-- ACC + MDR (4c₁), (4c₂) & (4c₃)
2. P-reg --> P-reg + 1 (5)
3. STATE --> FETCH



CPU STEPS DURING THE EXECUTION OF SMALL INSTRUCTIONS

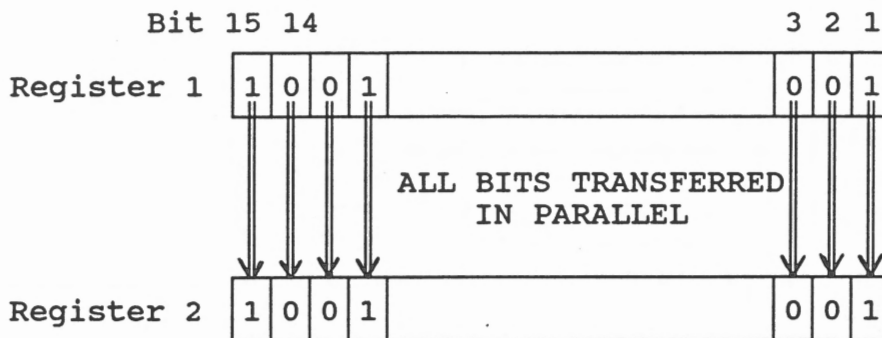
The table given below gives the steps taken in the execution of each of the basic instructions of SMALL.

OP CODE	STEPS
HALT	Computer stops. RUN flip-flop set to OFF
ADD	1) $I_{0-7} \rightarrow MAR_{0-7}, MAR_{8-15} = 0.$ 2) $M[MAR] \rightarrow MDR$ 3) $ACC \leftarrow ACC + MDR$
STORE	1) $I_{0-7} \rightarrow MAR_{0-7}, MAR_{8-15} = 0.$ 2) $ACC \rightarrow MDR$ 3) $MDR \rightarrow M[MAR]$
LOAD	1) $I_{0-7} \rightarrow MAR_{0-7}, MAR_{8-15} = 0.$ 2) $M[MAR] \rightarrow MDR$ 3) $MDR \rightarrow ACC$
JMP	$I_{0-7} \rightarrow P_{0-7}, P_{8-15} = 0.$
JEQ	IF $ACC = 0$ THEN $I_{0-7} \rightarrow P_{0-7}, P_{8-15} = 0.$ ELSE $P \rightarrow P + 1.$
JGT	IF $ACC > 0$ THEN $I_{0-7} \rightarrow P_{0-7}, P_{8-15} = 0.$ ELSE $P \rightarrow P + 1.$
JLT	IF $ACC < 0$ THEN $I_{0-7} \rightarrow P_{0-7}, P_{8-15} = 0.$ ELSE $P \rightarrow P + 1.$
SUB	1) $I_{0-7} \rightarrow MAR_{0-7}, MAR_{8-15} = 0.$ 2) $M[MAR] \rightarrow MDR$ 3) $ACC \leftarrow ACC - MDR$
INP	Keyboard value $\rightarrow ACC$
OUTP	$ACC \rightarrow$ Screen
INC	$ACC \rightarrow ACC + 1$
CALL s	1) $ACC \leftarrow P + 1$ (Address of next instr) 2) $I_{0-7} \rightarrow MAR$ (Address of 's') 3) $ACC \rightarrow MDR$ 4) $MDR \rightarrow M[MAR]$ s:=Return Address 5) $P \rightarrow I_{0-7} + 1$ (ie Jump to s+1)

check these micro steps out on
the assembler

DATA TRANSFER INSIDE THE COMPUTER

Data can be transferred between registers either serially or in parallel. Serial transfer is slow as each bit of the word is transferred one after another. For high speed each bit is transferred in parallel. This is illustrated below:



BUS STRUCTURE

There are a large number of registers in the CPU. To interconnect each register to every other one would lead to an exceptionally large number of wires. The problem is solved using a BUS structure.

A BUS consists of set of wires:

- 1 Wire for each bit (in our case 16 for parallel transfer)
- Wires for ADDRESS SPECIFICATION purposes
(if there are N registers on a bus then we need $\log_2 N$ wires ie 8 regs would require 3 wires).
- 1 Wire for SYNCHRONIZATION.
- 1 Wire for CONTROL
- 1 Wire for ERROR DETECTION

A BUS can potentially connect any register attached to it to any other register. At any one time however, data can only be transferred between two specific registers.

BUS STRUCTURE OF SMALL

Conceptually SMALL is similar to many micro-computers. It has 3 busses:

I/O bus for transfers between the I/O units and the ACCUMULATOR.

Memory bus for transfer of data from the CPU to Main memory.

Internal bus for transfers between the CPU registers and the ALU.(Some computers have more than 1 internal bus. Often 2.)

The SPEEDS of the busses differ (to match the physical characteristics of the device). The CPU bus is the fastest, and most expensive, while the I/O bus the slowest.

The BUS structure of SMALL is shown in Figure 6.2.

EXAMPLES:

- 1 Write an assembler program that reads in, sums & prints three values.
- 2 Write an assembler program that inputs two values. It outputs the larger of the two values.
- 3 Write an assembler program that inputs and prints all the positive integers given to it. A negative integer terminates the program. Print out the largest integer sent to it. (You can also print out the smallest integer).
- 4 Write an assembler program that inputs and outputs 20 values. The program must also print out how many of these values are ≥ 0 and how many are negative.
- 5 Modify Example4 to read in and print a discriminator value first and then print out how many values are \geq to it etc.

CHAPTER 7

LOOPS in ASSEMBLER

PREAMBLE TO CHAPTERS 7,8 & 9

The instructions available at the assembler level are very basic. There are a large number of constructs that we are accustomed to using in a High Level Language that now do not exist. A major theme of Chapters 7,8 & 9 will be the construction of these structures. They will then be used to solve appropriate problems. It is a basic theme of this text that good programming structures should be used at all times.

INTRODUCTION

The high level constructs of FOR, WHILE & REPEAT loops do not exist at the assembler level. These loops have to be constructed from first principles by the programmer. Test and Jump instructions are used to construct the various loops. Examples for all these loops are given.

FOR LOOP

In Pascal the FOR loop is written as follows:

FOR loop_var := start TO finish DO computation:

The underlying structure of this loop can be described as shown in Figure 7.1.

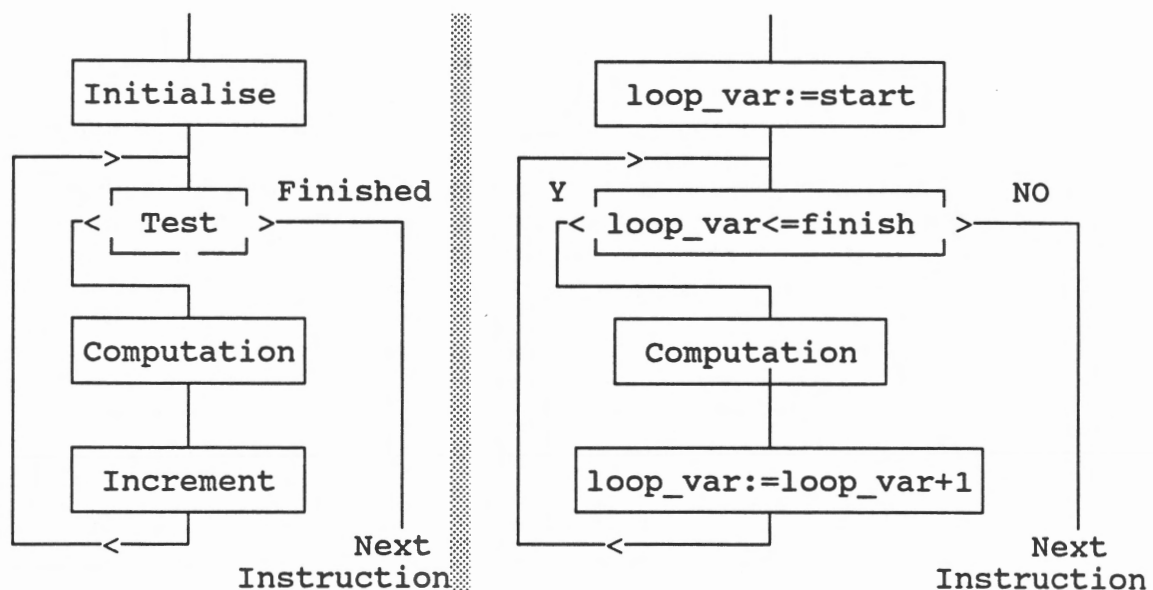


FIGURE 7.1 -- FOR LOOP STRUCTURE

The FOR loop is demonstrated in the following example which calculates & prints the sum of 1+2+3...+10. This is shown in Figure 7.2

	LOAD	init		;]	
	STORE	loop_var		;]	loop_var:=init
	LOAD	zero		;]	
	STORE	sum		;]	sum:=0
for_loop	LOAD	final			
	SUB	loop_var		;]	ACC:=finish-start
	JLT	stop		;]	Jump if complete
				;]	else continue
	LOAD	sum		;]	
	ADD	loop_var		;]	
	STORE	sum		;]	sum:=sum+loop_var
	LOAD	loop_var		;]	
	INC			;]	
	STORE	loop_var		;]	loop_var:=loop_var+1
	JMP	for_loop		;]	
stop	LOAD	sum			
	OUTP				
	HALT				
init	DEC	1			
final	DEC	10			
zero	DEC	0			
sum	BSS	1			
loop_var	BSS	1			
	END				

FIGURE 7.2

ASSEMBLER PROGRAMMING HINT: Validly the question can be put as to why I did not use the definition **sum DEC 0** and preferred **sum BSS 1** & then loaded zero into sum. If you are only going to run the program exactly once then there is no good reason. In practice, certainly while testing and often while running, the program is executed several times. In this situation the short-cut method gives problems as 'sum' will not be set to zero at the start of the program but will have the value left in it at the end of the previous run. In most cases this will lead to subsequent runs of the program giving the wrong answer.

WHILE LOOP

The structure of the WHILE is shown in Figure 7.3. The revised structure and the assembler code is shown in Figure 7.4.

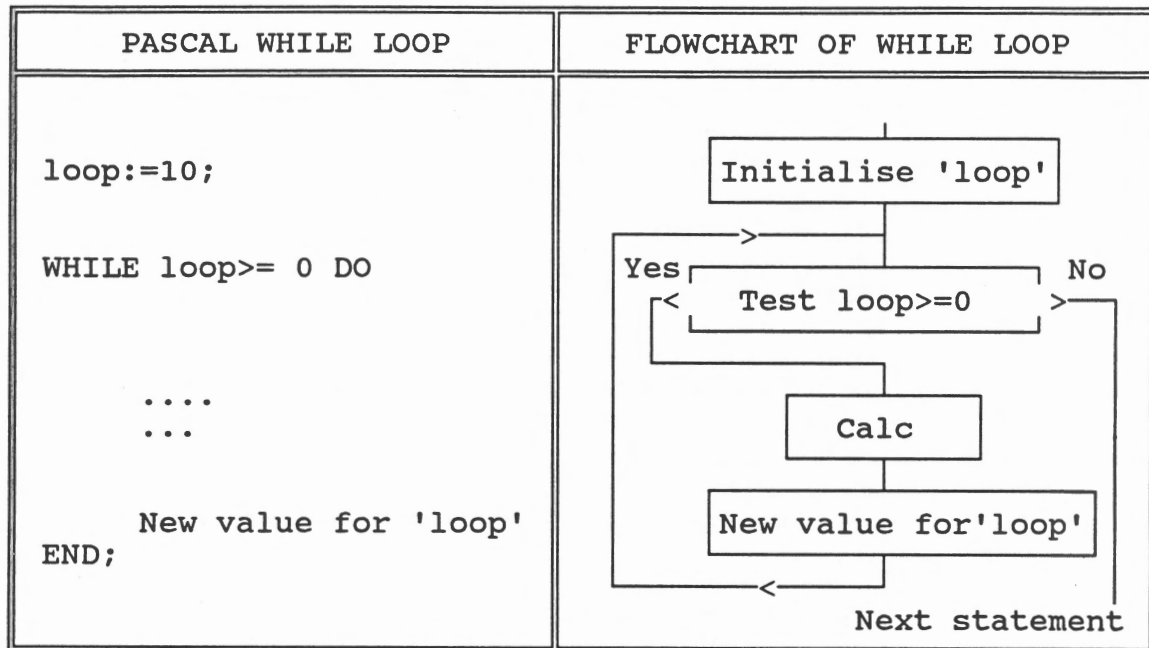


FIGURE 7.3

The test condition $\text{loop} \geq 0$ must be reversed to $\text{loop} < 0$ so that we jump out of the loop when this reversed condition is true and go through the loop when it is false.

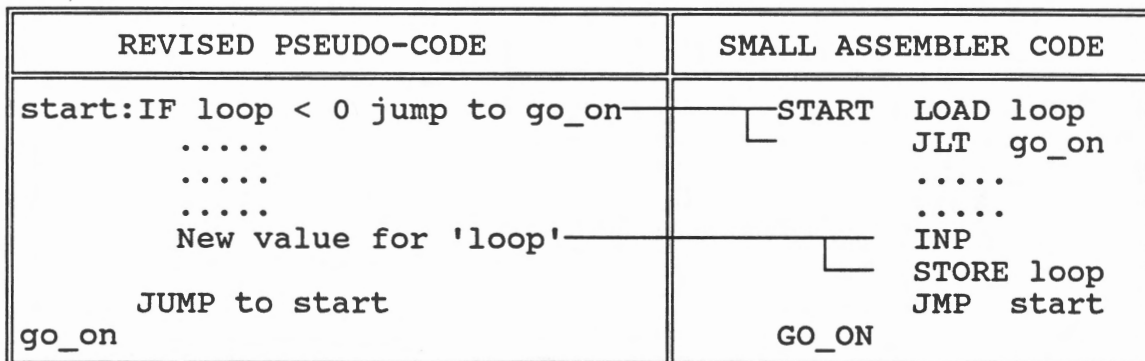


FIGURE 7.4

EXAMPLE 1

The following assembler program reads in and sums non-negative values. It also counts how many values are input. The number of values input and the sum of these values is output. The program terminates when a negative value is read. A WHILE loop construct is used.

LABEL	INSTRUCTION	COMMENT
	LOAD zero	
	STORE number	;number:=0
	STORE sum	;sum:=0
	INP	;Read a value
	STORE val	
start	LOAD val	
	JLT go_on	;jump if ACC<0
	LOAD sum	;Acc <-- sum
	ADD val	;Acc = Sum + count
	STORE sum	
	LOAD number	;
	INC	;
	STORE number	;number=number+1
	INP	
	STORE val	;READ(val)
	JMP start	
go_on	LOAD number	
	OUTP	;WRITE (number)
	LOAD sum	
	OUTP	;WRITE (sum)
	HALT	
zero	DEC 0	; Assign zero:=0
number	BSS 1	; Defines one store
val	BSS 1	
sum	BSS 1	
	END	

EXAMPLE2

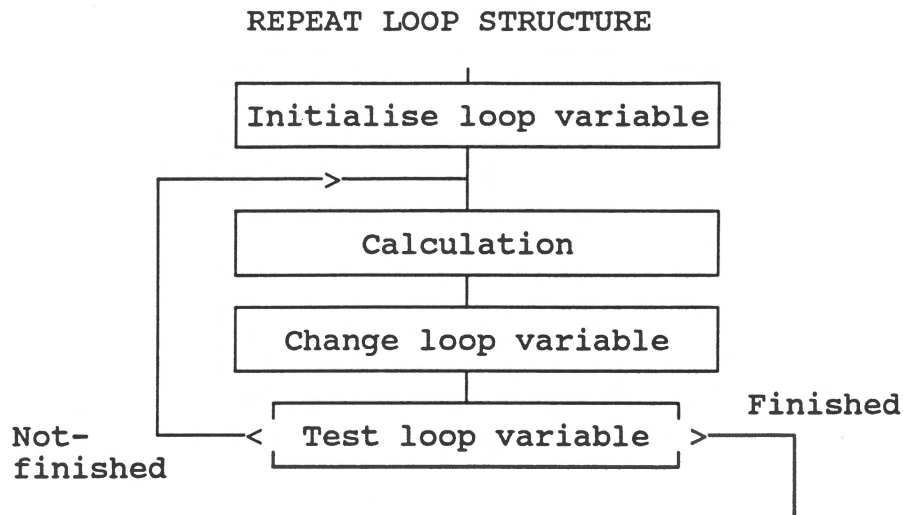
Read and sum values until a positive value is read in. The SMALL instruction set not have a JUMP >= command. Thus two tests have to be done. One to test > the other to test =. This is shown in Figure 7.5.

PASCAL	ASSEMBLER SKELETON	ASSEMBLER CODE
READ(val); WHILE val<0 DO BEGIN READ(val); END;	start IF val>=0 jump to go_on JMP start go_on	start LOAD val JGT go_on JEQ go_on JMP start go_on

FIGURE 7.5

REPEAT LOOP

The structure of the repeat loop is given as a flow chart.



The example below, shown in Figure 7.6, uses this basic 'repeat' structure. It Reads & Prints values. The program stops after the value 99 is handled. All the values are printed on one line unless there are too many in which case a second & subsequent lines are used. Essentially a Pascal WRITE statement is used.

repeat	INP	;ACC:= value read
	OUTP	;Print
	SUB n99	;ACC:=ACC-99
	JEQ stop	;if ACC=0 jump stop
	JMP repeat	;REPEAT
stop	HALT	
n99	DEC 99	
	END	

FIGURE 7.6

Should you only wish one integer value per line then the solution is shown in Figure 7.7.

repeat	INP	;ACC:= value read	
	OUTP	;Print	
	OUTCH cr	;Do a Carriage Return	—WRITELN
	OUTCH lf	;Do a Line Feed	
	SUB n99	;ACC:=ACC-99	
	JEQ stop	;if ACC=0 jump stop	
	JMP repeat	;REPEAT	
stop	HALT		
n99	DEC 99		
cr	DEC 13	;Binary 00001101	
lf	DEC 10	;Binary 00001010	
	END		

FIGURE 7.7

The Assembler analogy of a Pascal WRITELN statement is illustrated in Figure 7.7. The Carriage return character 'cr' causes the output pointer to return to the start of the line. The line feed character 'lf' causes the output device to move to the next line. (See Table 4.1 for the binary representation of these characters.)

EXAMPLES

- 1 Write an assembler program that reads in 2 integers A & B. ($B > A$). It then calculates the sum of the values (A, A+1, A+2, .. ,B) and prints this value. Use a FOR loop construct.
- 2 Write an assembler program that uses a FOR ... DOWNT0 loop that reads in 20 values and outputs those values that are less than 15.
- 3 Write an assembler program, using a WHILE loop, that reads and prints values until the value read is less than the previous value read.
- 4 Write an assembler program that uses a REPEAT loop to discard all negative values read in. The program is to read and sum all positive values and is terminated by the value 111 which must NOT be counted.

CHAPTER 8

ADDRESS MODIFICATION

INTRODUCTION

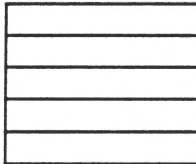
In this chapter the various ADDRESS MODIFICATION modes will be described. They are particularly useful when an array of values is to be accessed and used. For this reason ARRAYS are introduced first.

ARRAYS

Only one dimensional arrays can be defined in SMALL. An example is given below. The implementation 2-Dimensional arrays will be discussed later in the chapter.

Z BSS 5 ;Defines 5 stores

Z



The first store has the name Z. The other 4 stores have no name associated with them.

How do we access them?

AN INVALID ATTEMPT TO ACCESS EACH STORE OF THE ARRAY

Set all the 5 stores of array Z to zero.

```

LOAD      zero      ;ACC:=0
STORE     Z          ;Z:=0
STORE     Z+1
STORE     Z+2
STORE     Z+3
STORE     Z+4
HALT
zero DEC   0
Z  BSS    5          ; 5 stores defined
END
```

Unfortunately the address part of the instruction can not be an arithmetic expression. It can only be a store name. So this technique can not be used (Some more sophisticated assemblers do support this feature). This can be a very tedious method if a large array is involved.

In essence we require a method of MODIFYING THE ADDRESS. To do this an INDEX register is used. The value of the INDEX REGISTER is added to the operand address ie

$$\text{EFFECTIVE ADDRESS} = [\text{OPERAND ADDRESS} + \text{INDEX REG VALUE}]$$

INDEX REGISTERS IN SMALL

SMALL has 2 index registers called X & Y. A value in the Accumulator can be stored in an Index Register by a STORE instruction. A LOAD instruction moves the value in the Index Reg to the Accumulator. The INC instruction increments the Index Register. These are the only instructions that can be carried out on Index registers.

```
STORE @X ;moves the value in the ACC to the X reg
LOAD  @X ;moves the value of the X-Reg to the ACC
INC   @X ;Increments the Index Reg X by 1
```

The address modification method of accessing each element of an array is illustrated in Figure 8.1

ADDRESS MODIFICATION METHOD OF ACCESSING AN ARRAY

Set all the 5 stores of Z to zero.

	LOAD	zero	
	STORE	@X	;Index reg x:=0
start	LOAD	zero	;ACC:=0
	STORE	Z [@X]	;Store value in Acc in
			;store(Z+X)=(Z+0)=store Z
			;on next pass in store Z+1
			;then Z+2 etc)
	INC	@X	;Index Reg X incremented by 1
	LOAD	@X ?	
	SUB	five	;ACC:=ACC-5
	JEQ	go_on	;Jump to go_on if ACC=0
	JMP	start	
go_on	HALT		
zero	DEC	0	
five	DEC	5	
Z	BSS	5	
	END		

FIGURE 8.1

The essence of this method is to increase the EFFECTIVE ADDRESS by ONE each time the loop is traversed. This is done by adding ONE to the value of the Index-Reg each time. The BASE ADDRESS of the array stays unchanged throughout.

EXAMPLE 2:

Set the 5 stores of "S" to 1,2,3,4,5 respectively. The solution is shown in Figure 8.2.


	LOAD	zero	
	STORE	@X	;Index reg x:=0
start	LOAD	val	;ACC:=1
	STORE	S [@X]	;Store(S+X)=(S+0)=S<--ACC
	INC	@X	;@X:=@X+1
	LOAD	val	;  val:=val+1
	INC		
	STORE	val	
	SUB	six	;ACC:=val-6
	JEQ	go_on	;Jump to go_on if ACC=0
	JMP	start	
go_on	HALT		
zero	DEC	0	
val	DEC	1	
six	DEC	6	
S	BSS	5	
	END		

FIGURE 8.2

Note: In this solution the initial value of the store "val" is changed. Had we wanted to retain this value we could have made the definitions:

one	DEC	1
val	BSS	1

and inserted the extra code at the beginning of the program. Now the value of the store "one" is not destroyed.

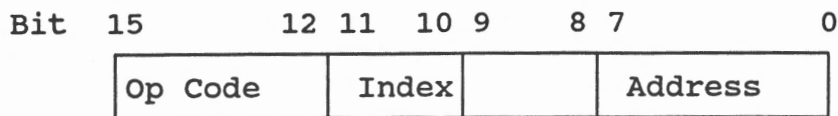
LOAD	one
STORE	val

IMPLICATIONS OF THE INTRODUCTION OF INDEX REGISTERS

In our basic machine representation of an instruction we need to specify:

- (a) whether indexing is to take place; and if so
- (b) which index register is to be used.

In SMALL the 2 bits, bits 11 & 10, are used to store this information. The meaning of the bit settings is given in Figure 8.3



INDEX BIT MEANINGS		
Bit 11	Bit 10	Meaning
0	0	No indexing
0	1	X Index register to be used
1	0	Y Index register to be used
1	1	Both X & Y Indx Regs are used

FIGURE 8.3

It is worth emphasising that a specific state indicating that NO indexing is to take place is essential.

Question: How many bits will be required if the machine has 4 index registers?

Answer: 3.

<u>Explanation:</u>	0 0 0	No indexing
	0 0 1	First Index Register
	0 1 0	Second Index Register
	0 1 1	Third Index Register
	1 0 0	Fourth Index Register
	1 0 1	Unused
	1 1 0	
	1 1 1	

In general N bits will allow the use of $(2^N - 1)$ index registers. The remaining state is used to notate that no indexing is indicated. 3 bits will support $2^3 - 1 = 7$ index registers.

INDIRECT ADDRESSING

To date we have used **DIRECT** addressing.

The Effective Address = Operand Address

For **INDIRECT** Addressing:

The Effective Address = CONTENTS of the Main Memory Location whose address is specified as the operand of the instruction

EXAMPLES

Main Memory

0	15
1	6
2	4
A 3	5
4	18
5	19
6	7

Type of Addressing	Instruction	Effective Address (EA)	Final value in Accumulator
Direct	LOAD A	EA=A=3	5
Indirect	LOAD [A]	EA=M[A]=M[3]=5	19

An **INDIRECT** address is like a **POINTER** in Pascal. It contains the address of the store required.

Indirect addressing is a useful technique of specifying the address of one store of an array. It is an alternate method to using an index register.

ENHANCED COMPUTER

Now that indirect addressing has been introduced our computer must be able to execute such instructions. This means that an extra phase, called the **INDIRECT** phase, must be added to the basic operation of the **CONTROL UNIT**.

The expanded actions of the Control Unit are shown in Figure 8.4.

```

FETCH:
  F1  MAR <-- P
  F2  MDR <-- M[MAR]
  F3  I   <-- MDR
  F4  MAR <-- Address part of I-Reg (bits 0-7)
  F5  IF INDIRECT Instruction THEN --> INDIRECT
      ELSE --> EXECUTE

INDIRECT
  I1  MDR <-- M[MAR]
  I2  MAR <-- MDR
  I3  IF INDIRECT bit of MDR set THEN --> INDIRECT
      ELSE --> EXECUTE

EXECUTE
  E1  Execute instruction
  E2  P   <-- P+1 (except for JUMP instructions)
  E3  FETCH

```

FIGURE 8.4

EXAMPLE:

In the Figures 8.5 (a,b,c) the detailed execution action of the following instruction, which is stored in word 7 of Main Memory, is traced:

LOAD [S] => LOAD [1] => LOAD 3. The value 19 is finally loaded to the Accumulator

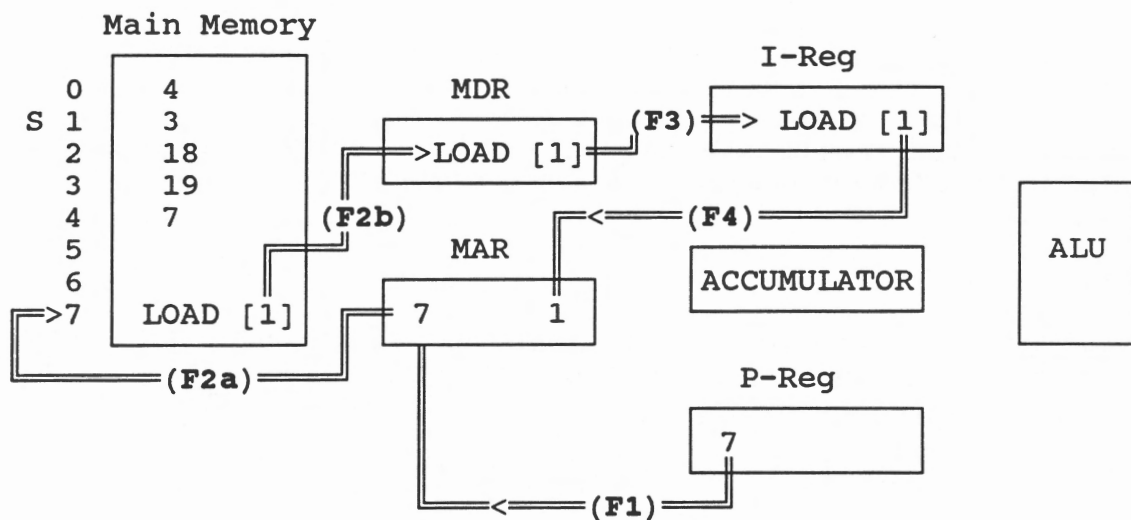
FETCH (INSTRUCTION)

FIGURE 8.5(A)

INDIRECT PHASE:

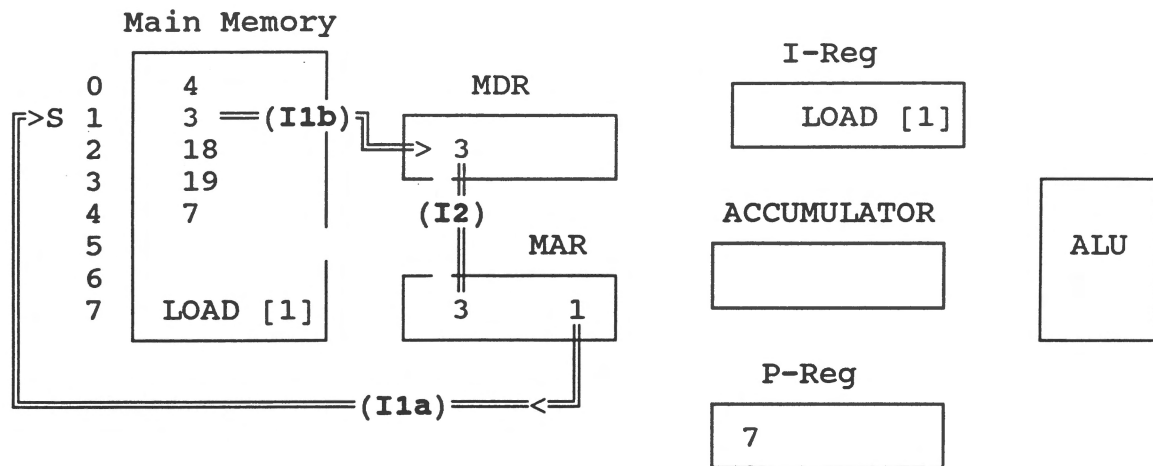


FIGURE 8.5(B)

EXECUTE (INSTRUCTION)

1. Instruction in I-reg is executed
 (It is a LOAD [1] = LOAD 3 instruction)
 The address 3 is already in the MAR (Indirect Phase)
 MDR \rightarrow M[MAR] (E1a) & (E1b)
 MDR \rightarrow ACC (E1c)
2. P-reg \rightarrow P-reg + 1 (E2)
3. STATE \rightarrow FETCH

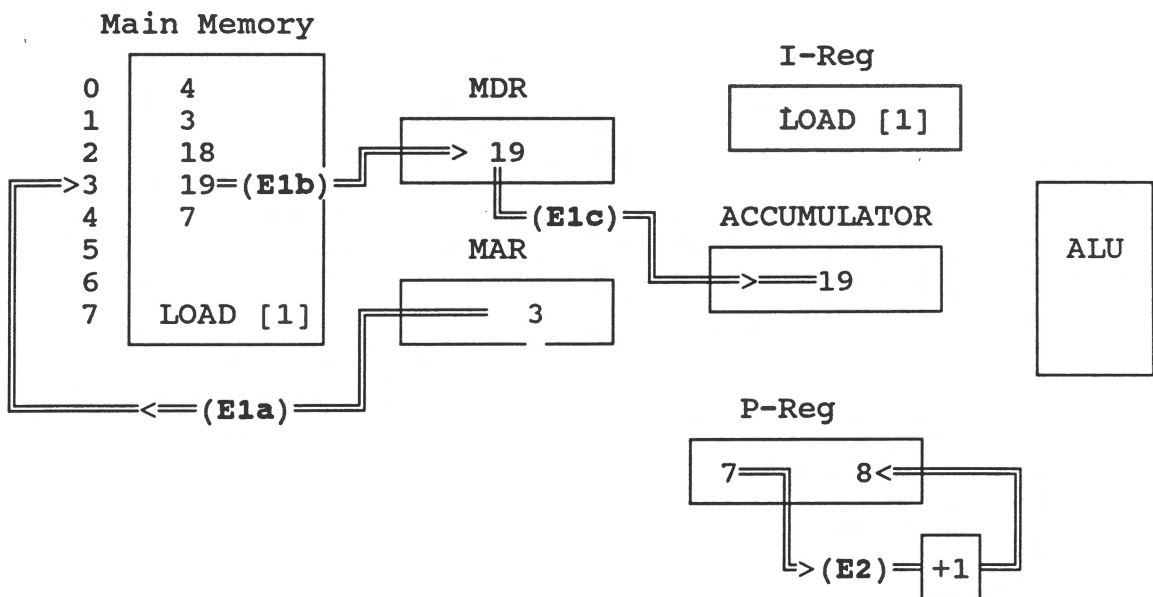


FIGURE 8.5(C)

ADDRESS DEFINITION STATEMENT

It is now necessary to have a statement which puts the address of a designated store into another store ie.

R ADDR S

This means: Put the address of S into R. In the example given below R is the store 25, S the store 15 and the ADDR statement sets the contents (or value) of store R to 15.

EXAMPLE

Use Indirect addressing to set the 10 successive stores of S to the values 0,1,2... ,9 respectively.

Main Memory

Address

0	LOAD	r	; [
1	STORE	addrx	;] Copies r to addrx.
2	start	LOAD	val
			; ACC=0 initially. Goes up by 1
			; for each iteration of loop.
3	STORE	[addrx]	; ACC--> S
			; (S+1, .. S+9) for
			subsequent iterations
4	LOAD	addrx	; [
5	INC		;] Increment ADDR X
6	STORE	addrx	;]
7	LOAD	val	; [
8	INC		;] Increment VAL
9	STORE	val	;]
10	SUB	ten	
11	JEQ	stop	; Quit. Problem finished.
12	JMP	start	; Do loop again
13	stop	HALT	
14	val	DEC 0	
15	s	BSS 10	
25	r	ADDR s	; Puts address of s into r
26	addrx	BSS 1	
27	ten	DEC 10	
		END	

INDIRECT ADDRESSING WITH INDEX REGISTER MODIFICATION

INTRODUCTION

It is possible to specify instructions that are INDIRECT and use an INDEX register as well. The question that arises is: Which is done first the indirection or the indexing? The answer is: it depends on how the machine is defined. Either can be implemented.

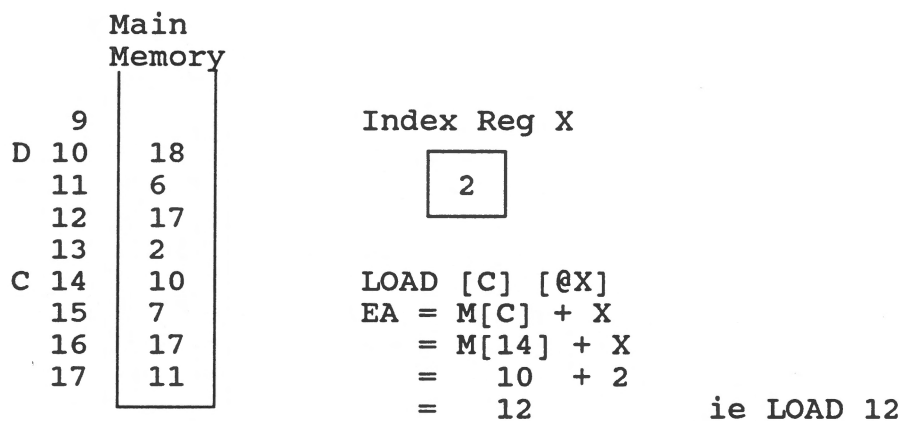
POST-MODIFICATION

INDIRECTION is done first then the INDEX register value is added.

LOAD [C] [@X]

Effective address (EA) = M[C] + X

EXAMPLE:



ACC = 17 after this instruction is executed

FIGURE 8.6

This type of addressing is useful for any array example. The start address of the array is used as the indirect element and the index register value provides the offset from that start address. In the above example D could have been defined as an array of 4 stores and C would have been defined to hold the address of the first store of array D ie

```
D    BSS    4
C    ADDR   D
```

LD [C] [@X]
 = LD [10] [@X]
 = LD 18 + [X] if X = 2
 = LD 20

PRE-MODIFICATION

Small does not support pre-modification. If it did it would work as follows: First, the value in the index register would be added to the operand address. Then only would indirection take place on this address.

For `LOAD [C] [@X]` Effective address (EA) = $M[C+X]$

This type of addressing can be beneficially used when a table (array) of addresses is being kept and you wish to specify the first address of a specific table.

The example in Figure 8.6 is redone using PRE-MODIFICATION.

`LOAD [C][X] = LOAD [14][2] = LOAD[14+2] = LOAD[16] = LOAD 17`

ACC = 11 after this pre-modified instruction was executed.

EXAMPLES OF INDIRECT ADDRESSING + INDEX REGISTER (POST-MODIFICATION)

Example 1: Set the values of 10 successive stores to 0,1,2 ... ,9 respectively.

Main Memory
Address

0	LOAD	val	;ACC=0
1	STORE	@x	;x=0
2 start	LOAD	@x	;ACC=0 initially, inc by 1 for
			;each loop iteration
3	STORE	[r] [@x]	;Put 0 in store r+0
			;For subsequent iterations
			;Put x in store r+x
4	INC		;ACC = ACC+1
5	STORE	@x	;@x=ACC
6	SUB	ten	
7	JEQ	stop	; Quit. Problem finished.
8	JMP	start	; Do loop again
9 stop	HALT		
10 val	DEC	0	
11 ten	DEC	10	
12 array	BSS	10	
22 r	ADDR	array	; Puts address of array into r
	END		

12	60
13	151
14	21
15	20
16	23
17	142
18	14
19	22
20	123
21	656
22	28
23	51
24	19

`LD [A] [@x]` $x \text{ Reg} = 1$
 $EA = [14] + 1$
 $= [21] + 1$
 $= 22 \quad A = 28$

`LD [A] [@x]`
 $EA = [14 + 1] = [15]$ Pre modification
 $= 20 \quad A = 123$

Computer only
supports one
type

Forget about
this one

Example 2: Read in the account number and the bank balance for 10 people. Then read in an account number and output both the account number and the bank balance. The account numbers will be kept in one array and the bank balances in another. For the lookup, the position of the account number in the array is found and put in an index register. This value is then used with the start address of the bank balance array to get the appropriate value. The solution is given in Figure 8.7.

```

; LOAD THE 10 ACCOUNT NUMBERS AND BANK BALANCES
0      LOAD      zero
1      STORE     @x          ;x=0
2  start INP
3      STORE     [st_no][@x] ; Input & store number
4      INP
5      STORE     [st_bal][@x] ; Input & store balance
6      LOAD      @x
7      INC
8      STORE     @x          ; x=x+1
9      SUB       ten
10     JEQ       go_on
11     JMP       start
; INPUT ACCOUNT NUMBER TO BE SEARCHED FOR
12 go_on LOAD      zero
13     STORE     @x
14     INP
15     STORE     val
16 search LOAD      val          ; search for account no
17     SUB       [st_no][@x]
18     JEQ       found
19     LOAD      @x
20     INC
21     STORE     @x
22     JMP       search
23 found LOAD [st_no][@x]          ; Account found values
24     OUTP          ; printed
25     LOAD [st_bal][@x]
26     OUTP
27     HALT
28 no    BSS     10
38 bal   BSS     10
39 st_no ADDR    no
40 st_bal ADDR    bal
41 ten   DEC     10
42 zero  DEC     0
43 val   BSS     1
44      END

```

FIGURE 8.7

Note: This solution is unsafe and will crash if an unknown bank number is searched for. To obviate this problem one should test, in the search loop, that index reg. x does not exceed 10. If it does you should jump to an error routine.

2-DIMENSIONAL ARRAYS

Only 1-Dimensional arrays are supported in Small. There are 2 methods of implementing 2-Dimensional arrays:

(1) Use a 1-dimensional array for each column of the array.

This method was illustrated in Figure 8.6. A 1-dimensional array was used for the account number and another 1-dimensional array was used for the bank-balance. This is a simple and easy method to use. The data definitions are given below.

```
no BSS 10
bal BSS 10
```

(2) Use one 1-Dimensional array large enough for all the values.

Here we could define the array as follows:

```
no_&_bal BSS 20
```

We could store the data in either of the following ways which are shown in Figure 8.8(A) & (B) respectively.

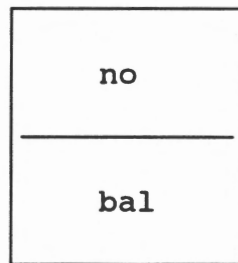


FIGURE 8.8(A)

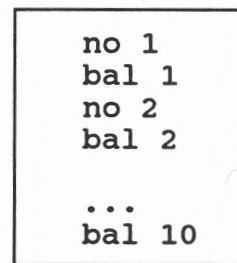


FIGURE 8.8(B)

In Figure 8.8(A) the 10 account numbers are followed by the 10 balances. The appropriate balance is 10 positions ahead of its account number.

Alternatively in Figure 8.8(B) the data was stored no1, bal, no2, bal, etc. In this case the appropriate balance is in the store immediately after the account number.

At the end of the day it is up to the writer of the program to decide how they wish to implement a 2-dimensional array (or a higher order array) and hence to determine the mapping function to use when storing and retrieving data in/from the array.

IMPLICATIONS OF INTRODUCING INDIRECT ADDRESSING

These different types of addressing have to be specified. 2 bits are necessary and bits 8-9 are used. The various states are shown below in Figure 8.8.

Bit 15 12 11 10 9 8 7 0

Op Code	Index	Mode	Address
---------	-------	------	---------

ADDRESSING MODE		
Bit 9	Bit 8	Meaning
0	1	Not used
0	0	Direct addressing
1	0	Indirect addressing
1	1	Immediate addressing

FIGURE 8.8

IMMEDIATE INSTRUCTIONS

In an IMMEDIATE instruction the VALUE to be manipulated is stored in the address part of the instruction.

Example:

Load the value 8 to the accumulator
The bit pattern of the word is shown below

Bit 15 12 11 10 9 8 7 0

Op Code	Index	Mode	Address
0100	00	11	00001000

IMMEDIATE instructions exist in small. The writer of the program can not specify such an instruction. This type of instruction is generated by the assembler.

SUMMARY OF ADDRESSING MODES

A real computer will support either PRE-modification or POST-modification but not both types. Of course it is possible to design a computer implementing both types. If this was done then it would be necessary to specify, in the assembler instruction, which type of modification was to be used. It would also mean that a specific bit would have to be set aside in the instruction word to hold this information. Most computers use POST-modification.

Given:

Main Memory
Address Value

Q	0	1
	1	3
	2	4
	3	6
	4	7
	5	8
	6	14
	7	21
	8	15

Index
Register X

3

Type of Addressing	Instruction	Effective Address (EA)	Final value in Accumulator
Direct	LOAD Q	$EA=Q=2$	4
Indexed	LOAD Q[@X]	$EA=Q+X=2+3=5$	8
Indirect	LOAD [Q]	$EA=M[Q]=M[2]=4$	7
Indirect + Indexing (Post-Modification)	LOAD [Q][@X]	$EA=M[Q]+X$ $=M[2]+3$ $=4+3=7$	21
Indirect + Indexing (Pre-Modification)	LOAD [Q][@X]	$EA=M[Q+X]$ $=M[2+3]$ $=M[5]=8$	15
Immediate	LOAD 8	Value in Address field	8

EXAMPLES

1 Given:

Main Memory		
Address	Value	
0	2	
1	0	
2	5	
3	6	
4	4	
P 5	1	
6	8	
7	3	
8	9	

Index
Register X

2

Work out the effective address and the final value in the accumulator for each type of addressing mode for the LOAD P instruction.

2 Write an assembler program that reads in values $0 < x < 11$ & counts the number of occurrences of each number. The program is terminated by a negative value & then prints out the results.

3 Write an assembler program to read in up to 20 positive numbers. Use a negative to terminate. Then read in a barrier value & print out all the values in the array larger than this barrier value.

CHAPTER 9

PROCEDURES

There are two types of procedures -- Open & Closed.

An Open procedure is commonly known as a MACRO. In essence the entire code of the macro replaces the macro name where-ever it appears in the program.

Closed procedures are the type that we are accustomed to from Pascal.

MACRO'S (OPEN PROCEDURE)

Small does not support MACRO'S. A Macro is an instruction that causes several instructions to be generated in its place. The MACRO is first defined. It has a name and parameters are usually supported.

MACRO DEFINITION

```
MACRO <name> <parameters>
    .....
    .....
    .....
END_MACRO
```

} Code statements

Wherever the MACRO name occurs in the code it is replaced by ALL the code of the macro.

EXAMPLE

SOURCE PROGRAM

```
MACRO double p1,p2
    LOAD  p1
    ADD   p1
    STORE p2
END_MACRO
```

ASSEMBLED PROGRAM

```

....
double t,z-----
....-----
                                LOAD  t
                                ADD   t
                                STORE z
                                >....
double s,p-----
....-----
                                LOAD  s
                                ADD   s
                                STORE p
                                >....
```


The advantage of this method is **SPEED** of execution and the **EASE** with which a macro can be **CHANGED**. The disadvantage is that a much **LONGER** program is produced.

PROCEDURES (CLOSED)

Only one copy of the code of the procedure exists. It can be called as many times as is required from within the program. The call to a procedure is in effect a jump to the first statement of that procedure. At the end of the procedure a jump back to the statement immediately after the one that called the procedure is executed.

Three new statements need to be introduced into **SMALL** so that procedures can be defined, entered, and exited.

PROCEDURE DEFINITION STATEMENT: **name PROC**

PROCEDURE ENTRY STATEMENT: **CALL name**

PROCEDURE RETURN STATEMENT: **RET [name]**

The details of how these statements work is explained in conjunction with the example given below.

EXAMPLE:

ASSEMBLER STATEMENTS		EXPLANATION
4	
5	CALL sum	1 The address of the next instruction is stored in the store called 'sum'. 2 Jump to address 'sum+1'.
6	next	
7	
8	
9	HALT	
10	sum PROC	Defines an empty store called 'sum' which is used to store the return address. After the CALL has been executed it will have, in this case, the value 6.
11	LOAD e	
12	ADD f	
13	STORE g	
14	RET [sum]	>Jump Indirect via 'sum'. ie Jump to the statement immediately after the one that called the procedure

PARAMETERS

Procedures by themselves are only part of the solution. For generality it is essential to be able to pass parameters to a procedure. In the following sections call by value, value-return & reference will be illustrated.

VALUE PARAMETERS

The CALL by VALUE technique involves the definition of a new store in the procedure. This store is initialised to the value of the parameter immediately before the procedure is called. Nothing is passed back to the calling sequence.

CALL by VALUE-RETURN is the same as above except that immediately after the procedure has been completed, the value of the variable, in the procedure, is copied to the variable in the calling sequence.

Example: Calculate and print the double of two values. A procedure is used. The value that is the input is passed by VALUE and the result calculated is returned by VALUE-RETURN.

```

0          INP
1          STORE a           ; Value in 'a'
2          STORE w           ; loaded to proc. var. 'w'
3          CALL double
4          LOAD ans           ; Result in proc. var 'ans'
5          STORE d           ; loaded to prog. var 'd'
                        ; for future possible use
6          OUTP

7          INP
8          STORE b
9          STORE w           ; Put next val in proc variable
10         CALL double
11         LOAD ans          ; Result in proc. var 'ans'
12         STORE e          ; loaded to prog. var 'e'

13         OUTP
14         HALT

15 double PROC
16         LOAD w
17         ADD w
18         STORE ans
19         JMP pr_end        ; Jump over vars of the proc.
20 w       BSS 1             ; Variables of the procedure
21 ans     BSS 1
22 pr_end RET [double]

23 a       BSS 1             ; Variables of the main prog
24 b       BSS 1
25 d       BSS 1
26 e       BSS 1
27         END

```

The above example illustrates several points:

1 Parameters are used. This enables a user to send different 'variables' to a procedure at different times. In our example first 'a' is sent to the procedure then 'b' is sent.

2 The variables 'w' & 'ans' were defined inside the procedure. This gives the illusion that they are local to the procedure however all variables defined in a program are visible to the entire program.

3 The names of the variables of the procedure are known to the main program. Before entry to the procedure the input values are loaded to the 'local variables' of the procedure. After exit from the procedure the result(s) are copied from the variable(s) of the procedure to the variable(s) of the calling program.

This copying can not be done inside the procedure because the name(s) of the appropriate variables of the main program are not known to the procedure.

REFERENCE PARAMETERS

The same problem as before is redone using CALL by REFERENCE. In essence the ADDRESS of the store concerned is passed to the procedure. Because all variables are GLOBAL in assembler there is little difference between using call by value or call by reference. You gain no extra security by using call-by-value as you did in Pascal.

Example: Calculate and print the double of a value. A procedure is used. The input and output value is passed to the procedure using CALL-by-REFERENCE.

```

0          INP
1          STORE a
2          LOAD aa          ; □ Address of 'a'(input parameter)
3          STORE ww          ; □ is stored in 'ww' in the proc.
4          LOAD dd          ; □ The address 'dd'(for the result)
5          STORE ans          ; □ is stored in 'ans' in the proc.
6          CALL double
7          LOAD d          ; Restore the result
8          OUTP
9          HALT
10 double PROC
11          LOAD [ww]          ; □ Note that all the addresses
12          ADD [ww]          ; □ in the procedure are now
13          STORE [ans]          ; □ INDIRECT
14          JMP pr_end
15 ww      BSS 1
16 ans     BSS 1
17 pr_end  RET [double]
18 a       BSS 1
19 aa      ADDR a
20 d       BSS 1
21 dd      ADDR d
          END

```

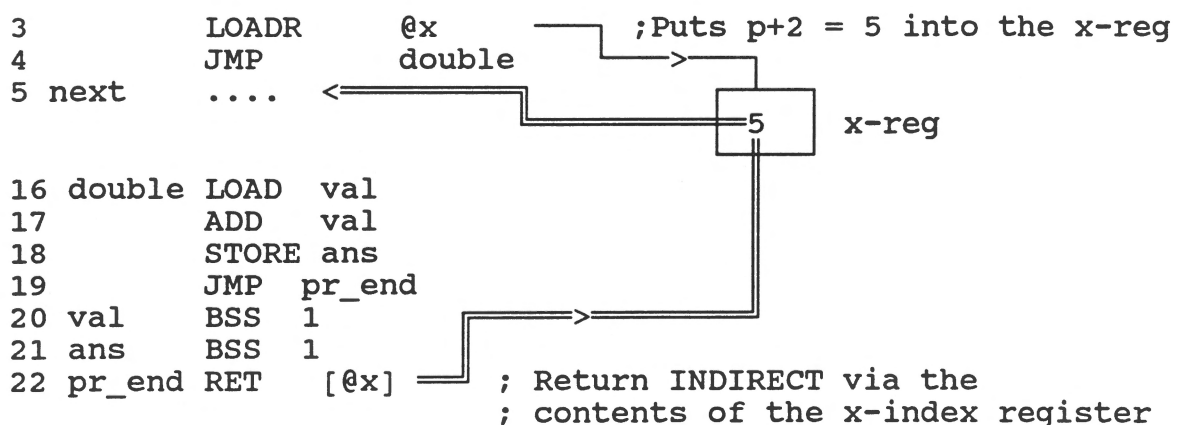
1 All the address information was done before the procedure was entered.

3 The data stores associated with each procedures were defined at the end of that procedure. A jump around them was performed. This is the preferred technique. As all the data definitions in assembler are global these stores could have been defined at the end of the program with all the other data definitions. However putting all the definitions at the end means a very long list of definitions and it is difficult to know which stores belong to which procedures.

Two other methods are in popular use. Neither of these methods is supported in SMALL. These methods will be described here for the reader's information. They are:

- (a) Use of a Register of the CPU to store the return address. In SMALL an Index register would have to be used.
- (b) Use of a STACK of CPU Registers to store return addresses.

In this example an index register is used to store the return address. A new instruction **LOADR** is needed. This instruction puts, into the x-register, the value of the p-register + 2. Then a jump to the start of the procedure is executed. At the end of the procedure a return, indirect, via the contents of the index register is performed.



This technique is only partially satisfactory. The reason is that the index register used for the return address can not be used for anything else (unless its value is specifically stored by the user program.)

STACK OF CPU REGISTERS

To overcome the short-comings of the previous methods a STACK of CPU registers can be used to keep the return address(es) for use on exit from a procedure.

The advantages of this method are threefold: (a) an index register is not used and thus unavailable for other purposes; (b) a procedure can now call other procedures from inside of itself (Note that this was possible in SMALL because an index register was not used to keep the return address); and (c) RECURSIVE calls are now possible.

Two new operations, PUSH & POP, need to be defined. They are used in conjunction with the STACK of CPU registers that keep the return addresses and an extra CPU register called TOP that indicates the specific STACK register to be used.

PUSH:	1	Puts the RETURN address into the register of the CPU stack indicated by TOP.
	2	TOP = TOP + 1
	3	JUMP to start of PROCEDURE.
POP	1	RETURN INDIRECT via the CPU stack register indicated by TOP.
	2	TOP = TOP - 1

Example: A value is quadrupled by doubling the value twice

```

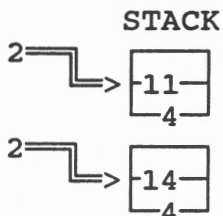
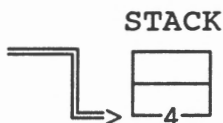
0      INP                ; Initially TOP=0
1      STORE a
2      STORE q
3      PUSH quad          ; TOP=1
4      LOAD q_ans          ;
5      STORE ans           ;
6      OUTP                ;
7      HALT

8 quad  LOAD q
9      STORE d             ;
10     PUSH double        ; TOP = 2
11     LOAD d_ans         ;
12     STORE d            ;
13     PUSH double        ; TOP = 2
14     LOAD d_ans         ;
15     STORE q_ans        ;
16     JMP e_quad
17 q     BSS 1
18 q_ans BSS 1
19 e_quad POP              ; TOP = TOP - 1
                                ; Top --> 0 In this example

20 double LOAD d
21     ADD d
22     STORE d_ans
23     JMP e_double
24 d     BSS 1
25 d_ans BSS 1
26 e_double POP           ; TOP = TOP - 1
                                ; After 1st call Top --> 1
                                ; After 2nd call Top --> 1

27 a     BSS 1
28 ans   BSS 1
      END

```



The STACK structure allows for the use of RECURSION (the ability for a procedure to call itself). Certainly the call to itself and the return is neatly done. Remember that it is up to the writer of the recursive routine to make a copy of the LOCAL variables of the procedure before any such call and to retrieve this copy on return to the same level of the procedure (again a stack is a useful mechanism to use).

In this description the STACK has been implemented as a number of CPU registers. We have assumed that enough registers exist for all purposes. Practically there is however some limit to the number of registers. To overcome this limitation the STACK is often implemented, not as CPU registers, but as locations in Main Memory. This will be slightly slower but the STACK is then effectively unlimited in size.

ADDRESS MODIFICATION -- AWFUL METHOD OF RETURN FROM A PROC.

Before jumping to the start of a procedure you **MODIFY** the last instruction of that procedure to jump back to the statement after the call to the procedure.

ADDRESS MODIFICATION METHOD

```

0          LOAD  rtn_1
1          STORE end_d
2          JMP   double

3          LOAD  rtn_2
4          STORE end_d
5          JMP   double
6          HALT

7 double   LOAD  val
8          ADD   val
9          STORE ans
10 end_d    BSS 1           ; This defines a blank store.

11 rtn_1    BINARY 0101000000000011 ; JMP 3
12 rtn_2    BINARY 0101000000000110 ; JMP 6
13
14

```

NOTE : Only stores related to the **ADDRESS MODIFICATION** have been defined. Data passed to/from the procedure has been omitted.

WHY is this method considered to be AWFUL?

- (1) The hassle to calculate the binary equivalent of the assembler statement **JMP x**
- (2) Should you add even one extra instruction to your program the return address has to be changed. Its easy to forget to do this or to get the change wrong. In the above example if an extra instruction was added at the beginning of the program the following 2 changes would have to be made:

```

0          LOAD  rtn_1
1          STORE new           ; New added instruction
2          STORE end_d
3          JMP   double
4          LOAD  rtn_2

          .....

13 rtn_1    BINARY 0101000000000100 ; JMP 4    **ADDRESSES**
14 rtn_2    BINARY 0101000000000111 ; JMP 7    ** CHANGED **

```

EXAMPLES

- 1 Write a procedure that accepts one value as input. This value specifies how many integers the procedure is to read, sum & print the total of. Use this procedure in a program that calls it from a loop.
- 2 Write a procedure that reads in values until it has received 3 values that have increasing values. It then prints these values. (Input 3 2 7 6 5 8 and outputs 3 7 8).
- 3 Write an assembler procedure to multiply two values together. Use repeated addition. Test that your procedure works.
- 4 Write a procedure SWOP that swops two values if the first value is larger than the second. Use this procedure to do the first sweep of a bubble sort. Print out the largest value in the array. Implement a full bubble sort.

CHAPTER 10

CHARACTERS

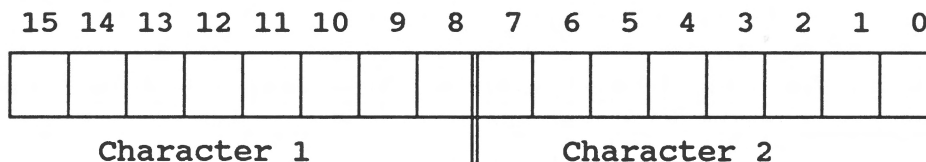
INTRODUCTION

Characters and the instructions needed to manipulate them were introduced into SMALL for the first time in 1990. Some modifications were introduced in 1991. I shall outline the concepts and instructions that have been implemented.

CHARACTER REPRESENTATION

The 8-bit ASCII character code is used to represent characters. The bit pattern for each character is given in Table 4.1 (with $b_7=0$).

SMALL has a 16-bit word hence 2 characters can be stored in each word.



I/O INSTRUCTIONS

INPUT A CHARACTER INCH 1 char is input to bits 7-0 of the Accumulator

OUTPUT A CHARACTER OUTCH 1 char is output from bits 7-0 of the Acc.

There are two special characters **Carriage Return 'cr'** and **Line Feed 'lf'** that are used to control output.

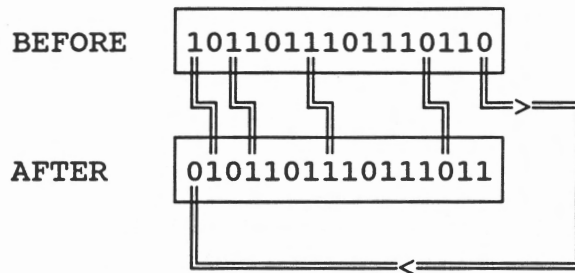
When the 'cr' character is output it causes the output pointer to return to the front of the output buffer.

When the 'lf' character is output it causes the output device to skip to the next line.

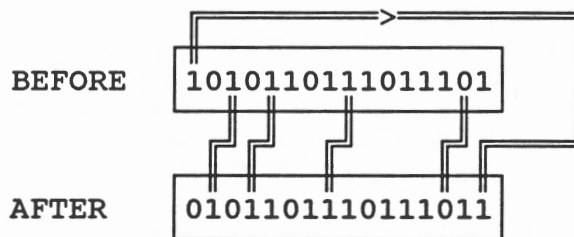
The action of an old-fashioned typewriter is a good example. The 'cr' returns the carriage to its home position at the left-hand side of the line. The 'lf' causes the paper to be shifted ('up' in practice).

ROTATE/SHIFT INSTRUCTIONS

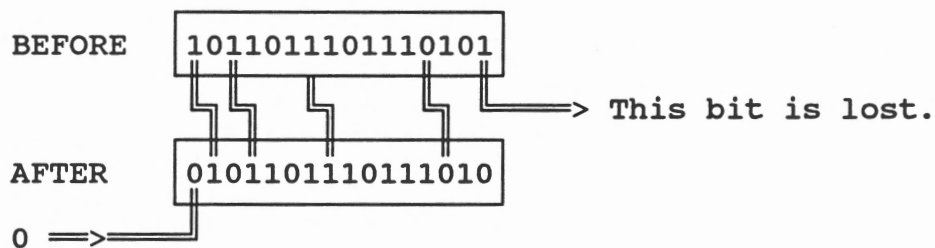
ROTATE-RIGHT ROR Rotate-right the Accumulator by 1 bit. (All the bits shift 1 to the right. The right-most bit rotates around to left-most bit.)



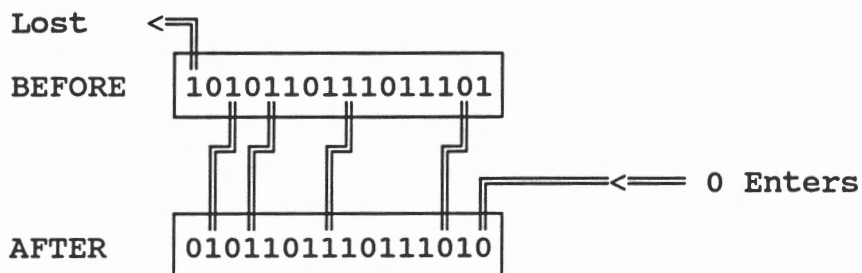
ROTATE-LEFT ROL Rotate-left the Accumulator by 1 bit. (All the bits shift 1 to the left. The left-most bit rotates around to the right-most bit.)



SHIFT-RIGHT SHR Shift-right the Accumulator by 1-bit. (All the bits shift 1 to the right. The right most bit is lost. A 0 enters as the left-most bit).



SHIFT-LEFT SHL Shift-left the Accumulator by 1-bit. (All the bits shift 1 to the left. The left-most bit is lost. A 0 enters as the right-most bit).



EXAMPLE: Read in a string of characters (<40) terminated by the character #. Store this character string in memory as well as printing it out. Finally, on the next output line, print out the number of characters read in.

```

go_on      LOAD      zero
           STORE     @X
           INCH
           OUTCH
           STORE     array[@X]
           INC      @X          ; Increment X Register
           SUB      hash        ; Test for # terminal char.
           JEQ      stop        ; If # was found
           JMP      go_on
stop        LOAD      cr
           OUTCH      ;Output pointer returned to front
           LOAD      lf
           OUTCH      ;Output device skips to next line
           LOAD      @X        ;@X keeps count of number of
                                ;characters read.
           OUTP
           HALT
zero        DEC      0
hash        DEC      35      ;Binary 00000000000100011
cr          DEC      13      ;Binary 00000000000001101
lf          DEC      10      ;Binary 00000000000001010
array       BSS      40
           END

```



See Table 4.1

Note : This solution:

- (a) stores only 1 character per word of memory (It is left as an exercise for the student to pack two characters per memory word).
- (b) counts, stores and prints the string terminal character # as part of the string.
- (c) This example has not been tested at the time of writing because the updates to SMALL have not as yet been completed.

DEFINING A CHARACTER STRING



Heading STR 'a b sum' Defines the string called Heading containing 'a b sum'. These 7 characters will be stored in 4 words. A trailing NUL character (00000000) is added because the string consists of an odd number of characters. Note that a **blank** character has the binary representation (01000000) and is different from the NUL character

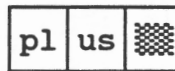
NUL = 
Blank = 



In the case where the defined Character String has an "even" number of characters two NUL characters are appended to the end of the string ie

Add_op Heading STR 'plus' Defines the string called Add_op containing 'plus'. These 4 characters will be stored in 2 words. Two trailing NUL characters (0000000000000000) are inserted into a 3rd word.

NUL = 
Blank = 



The strategy of placing a NUL character at the end of any defined string allows the programmer to easily detect the end of the string. This is necessary when the string is to be printed out as it is the programmers responsibility to move along the string printing out each character until a NUL is encountered.

BOOLEAN INSTRUCTIONS

AND **AND mask** The values in the Accumulator and the store 'mask' are ANDed together with the result being placed in the Accumulator. Note that a 1 is output only if both the bits are 1.

	BEFORE	AFTER
mask	1100001110000111	1100001110000111
ACC	1011110100110001	1000000100000001

OR **OR mask** The values in the Accumulator and the store 'mask' are ORed together with the result being placed in the Accumulator. Note that a 1 is output is either or both bits is a 1.

	BEFORE	AFTER
mask	1100001110000111	1100001110000111
ACC	1011110100110001	1111111110110111

XOR **XOR mask** The values in the Accumulator and the store 'mask' are XORed together with the result being placed in the Accumulator. Note that a 1 bit is output when exactly 1 of the two input bits is 1.)

	BEFORE	AFTER
mask	1100011100011001	1100011100011001
ACC	1000110100110001	0100101000101000

NOT **NOT** Each bit of the Accumulator is flipped

	BEFORE	AFTER
ACC	1001110000110101	0110001111001010

EXAMPLE: Print out the characters of a defined string.

```

                                zero
                                @X
start LOAD      finis [@X]
      ROR
      ROR
      ROR
      ROR
      ROR
      ROR
      ROR
      ROR
      OUTCH      ; 1st char is output
      LOAD      finis [@X] ; 2nd char is in bits 7-0
      OUTCH      ; 2nd char is output
      AND mask   ; Checks for a NUL as 2nd character
      JEQ stop   ; Stop if acc=0. This implies that
                  ; the 2nd char was a NUL.

      INC @X
      JMP start
zero DEC 0
mask DEC 255 32767 try this ; Binary 0000000011111111
finis STR 'End of chapter'
stop HALT
      END



```


8 rotates puts the 1st character in bits 7-0

read to check for as well

Some notes about this solution:

(a) The string finis would be stored as follows:

NUL = 
 Blank = 

En	d	of	c	ha	pt	er	
----	---	----	---	----	----	----	---

- (a) The stopping condition relies on the fact that there will be a NUL (00000000) in the 2nd character of the last word. **Acc AND mask** only gives 0000000000000000 if the 2nd character is NUL in which case we stop. If both the 1st & 2nd characters were NUL the first NUL would have been printed. You would not have seen anything because the NUL is printed as a blank.
- (b) Certainly there are several different ways of doing this example. The student is encouraged to investigate them.
- (c) This example has not been tested at the time of writing because the updates to SMALL have not as yet been completed.

EXAMPLES

1. Given the following string definition: **line1 STR 'First Line of Report'**. Write a SMALL program to print out the string {or any other string} (a) on one line, and (b) one character per line. Count and print (a) the number of non-NUL characters in the string (b) the number of NUL characters .

2. Write a SMALL program to calculate and print the number of words used to store any defined string (NUL's inclusive).

3. Write a SMALL program to calculate and print the number of occurrences of some specific character in a string.

4. Write a SMALL program that reads in 2 integers, sums them and produces the following output if the numbers were 7 & 4:

7 plus 4 gives the sum of 11

5. Write a SMALL program that reads in a string of characters (<30) and stores all the characters in memory. An integer to be used as an offset is then read in. Print out the string. Add the offset to each character of the string. Print out the new (encoded) string. How would you decode the string?

CHAPTER 11

MACHINES with DIFFERING NUMBERS of ADDRESSES

Small is an example of a 1-Address machine. In other words only ONE memory address is associated with each instruction. It is possible to design machines with 0,1,2,3 & 4 addresses associated with each instruction. Each of these machines will be described in turn. The same example will be used for each machine to illustrate its features and also for comparison purposes.

In this section the assumption will be made that 1 memory cycle is broken down into a number of sub-cycles. Also that there is sufficient time in 1 memory cycle for all the operations to complete. The execution of certain very long instructions (like Multiply & Divide) may take more than 1 memory cycle.

To load an instruction the following steps would be needed (where each requires a sub-cycle or some multiple of sub-cycles. Certain ops can be done in parallel):

```

1      P --> MAR
2,3    M[MAR] --> MDR
4      MDR --> I-Reg

```

To execute the instruction ADD X takes:

```

1      I0-7 --> MAR
2,3    M[MAR] --> MDR & (during 3) P --> P+1
4      ACC + MDR --> ACC

```

1-ADDRESS MACHINE

Small is an example of a 1-address machine. A sample instruction is:

```

LOAD a  ———>1 Memory cycle is used to fetch this
              instruction from Main Memory and put it in
              the I-Register.
              ———>1 Memory cycle is used to execute the
                    instruction. (For 'LOAD a' the value in store
                    'a' of Main Memory is moved to the
                    Accumulator).

```

For a 1-address instruction 2 memory cycles are required by the computer to execute an instruction.

Example: For the expression $A := B * (C + D * E - F / G)$ derive the 1-address code and analyse the implications.

```

LOAD   F
DIV    G
STORE  T1    ; T1:=F/G
LOAD   D
MPY    E      ; ACC=D*E
ADD    C      ; ACC=C+D*E
SUB    T1     ; ACC=C+D/E-F/G
MPY    B      ; ACC=ACC*B
STORE  A

```

9 Instructions
= 18 memory cycles

Op Code	Address Type	Operand Address
4 bits	4 bits	8 bits



16 bit word length
required

2-ADDRESS MACHINE

For 2-address instructions the operation is performed on the two operands and the result is placed in the first operand.

Operation A,B Means A \leftarrow A operation B
ie DIV C,D Means C \leftarrow C/D

Each instruction will take 4 memory cycles to execute. 1 memory cycle for each of the following: Fetch Instruction, Fetch Operand1, Fetch Operand2 & Perform Operation, and finally Store result in Operand 1.

Example: For the expression $A := B * (C + D * E - F / G)$ derive the 2-address code and analyse the implications.

```

MOVE   A,D    ;A=D
MPY    A,E    ;A=D*E
MOVE   T1,F   ;T1=F
DIV    T1,G   ;T1=F/G
SUB    A,T1   ;A=A-T1=D*E-F/G
ADD    A,C    ;A=A+C=C+D*E-F/G
MPY    A,B    ;A=A*B=B*(C+D*E-F/G)

```

7 Instructions
= 7*4
= 28 mem cycles

This solution does not destroy (overwrite) the value of any of the variables of the expression. By overwriting some variables a shorter solution can be obtained.

Op Code	Address Type	Op. Addr1	Op. Addr2
4 bits	4 bits	8 bits	8 bits



24 bit word
length required

As can be seen the length of the word increases and the number of memory cycles has also increased. 2-address machines are always the least efficient. The reason is that a large number of intermediate results are stored back to main memory with no use being made of these stored results.

3-ADDRESS MACHINE

For 3-address instructions the operation is performed on the first two operands and the result is placed in the third operand.

Operation B,C,A Means $A \leftarrow B \text{ operation } C$
 ie SUB B,C,A Means $A \leftarrow B - C$

Each instruction will take 4 memory cycles to execute. 1 memory cycle for each of the following: Fetch Instruction, Fetch Operand1, Fetch Operand2 & Perform Operation, and finally Store result in Operand 3.

Example: For the expression $A := B * (C + D * E - F / G)$ derive the 3-address code and analyse the implications.

MPY	D, E, T1	; T1 = D * E	} 5 Instructions = 5 * 4 = 20 mem cycles
ADD	C, T1, T1	; T1 = C + T1 = C + D * E	
DIV	F, G, T2	; T2 = F / G	
SUB	T1, T2, T1	; T1 = T1 - T2 = C + D * E - F / G	
MPY	B, T1, A	; A = B * (C + D * E - F / G)	

Op Code	Address	Op. Addr1	Op. Addr2	Op. Addr3
	Type			
4 bits	4 bits	8 bits	8 bits	8 bits

--	--	--	--	--

32 bit word length is required.

4-ADDRESS MACHINE

For 4-address instructions the operation is performed on the first two operands and the result is placed in the third operand. The 4th operand is used to specify the address of the next instruction in memory to be executed. With such instructions there is no need for a P-register in the CPU.

Operation B,C,A,Addr next ins Means $A \leftarrow B \text{ operation } C$
 ie SUB B,C,A,7 Means: $A \leftarrow B - C$ and next instruction is in word 7.

Each instruction will take 4 memory cycles to execute. 1 memory cycle for each of the following: Fetch Instruction, Fetch Operand1, Fetch Operand2 & Perform Operation, and finally Store result in Operand 3.

Example: For the expression $A := B * (C + D * E - F / G)$ derive the 4-address code and analyse the implications.

MPY	D, E, T1, 1	;T1=D*E	}	5 Instructions = 5*4 = 20 mem cycles
ADD	C, T1, T1, 2	;T1=C+T1=C+D*E		
DIV	F, G, T2, 3	;T2=F/G		
SUB	T1, T2, T1, 4	;T1=T1-T2=C+D*E-F/G		
MPY	B, T1, A, 5	;A:=B*(C+D*E-F/G)		

Op Code	Address Type	Op. Addr1	Op. Addr2	Op. Addr3	Addr of next inst
4 bits	4 bits	8 bits	8 bits	8 bits	8 bits

40 bit word length is required.

The 4-address machine was briefly used early on in the development of computers then it went out of fashion. The main reasons were the long word length required; together with the fact that a CPU register to hold the address of the next instruction was a very satisfactory solution.

Before entirely writing off such a machine let us consider a special application for it. Snobol is a high level language specifically set up to perform pattern matching problems. A typical statement in the language would be:

Compare 2 strings for equality, If equal goto 3 else goto 10

A suitable machine to handle this statement could be:

Op Code	Addr Type	Opernd1	Opernd2	Next Ins if Success	Next Ins if Failure
8 bits	4 bits	8 bits	8 bits	8 bits	8 bits
CMPEQ				3	10

The concept of keeping the address of the next instruction as part of the instruction can be a useful one.

0-ADDRESS MACHINE

A 0-address machine treats the memory of the computer as a STACK. The address of the data does not need to be specified. The data is IMPLICITLY assumed to be at the top of the stack (ie the top of memory). It is not possible to specify ALL instructions with 0-addresses, some instructions require an address. For these a 1-address instruction is used. So a 0-address machine uses both 0 & 1-address instructions.

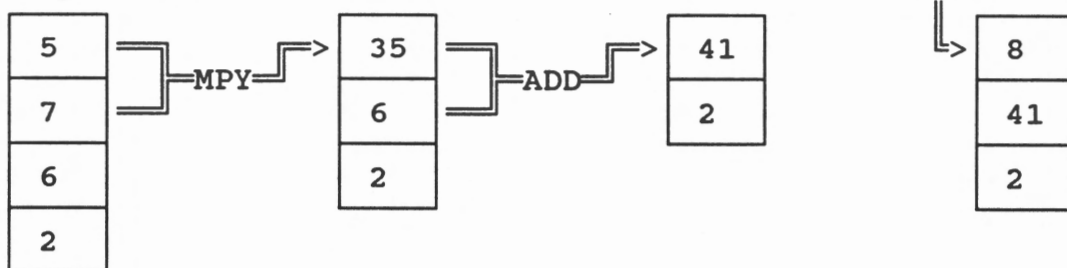
For 0-address instructions only 1 memory cycle is needed to execute an instruction. Usually there is special hardware in such a machine that treats the top 2 words of memory (ie the top of the stack) as CPU registers. Access to these words is thus at CPU speed and not at memory access speed.

For a 1-address instruction memory cycles are required by the computer to both fetch and execute the instruction.

Example: For the expression $A = B * (C + D * E - F / G)$ derive the 0-address code and analyse the implications. $\{A = 2 * (6 + 7 * 5 - 8 / 4)\}$

First the expression has to be converted to Reverse Polish Form: $BCDE* + FG/- := A$ then the code can be written. $\{2675* + 84/- := A\}$

Memory Stack



LOAD	B	
LOAD	C	
LOAD	D	
LOAD	E	
MPY		5
ADD		+ 7
LOAD	F	
LOAD	G	
DIV		
SUB		
MPY		
STORE	A	

0-Addr Instrs = 5*1 = 5 Mem cyc

1-Addr Instrs = 7*2 = 14 Mem cyc

19 Mem cyc

Op Code	Address	Operand Address
4 bits	Type(4Bits)	8 bits

8 bits needed for 0-addr.

16 bit word length
required for 1-addr.

A COMPARISON OF THE VARIOUS ADDRESSING SCHEMES

No. of Addresses	No. of Instr.	No. of Accesses / Instr.	Total No. of Memory Accesses	Word size (Bits)
4	5	4	20	40
3	5	4	20	32
2	6	4	24	24
1	9	2	18	16
0	12		19	
----- ----->1 ----->0	----- 7 5	----- 2 1	----- 14 5	----- 16 8

So what can be concluded?

(a) 2-address machines are always worst. This is because a lot of results are automatically stored back in memory for no useful purpose.

(b) There is not a lot to choose between the other machines as far as number of memory accesses are concerned.

(c) 3 & 4-address machines are less popular because of the long word-length required. (Special purpose machines excluded). It is also worthy of mention that for real machines 4 bits for the op.code is very small (giving 16 instructions only). More commonly 8 bits (256 instr.) would be used.

(d) 1-address machines are the most popular although there is little to choose between the 0 & 1-address machines.

I would point out that it is dangerous to take one example and to generalise from its result. I can only assure the reader that, from experience, this example is typical.

EXAMPLES

1 For each of the expressions $A := B * ((C * D) / (E - F)) + G$ & $X := S / T + A * (U - V) - U / V$ calculate how many instructions and how many memory accesses are needed for each of 0, 1, 2 & 3 address machines.

2 For a machine with 110 instructions, 8 index registers and the need to address 1048575 addresses calculate how many bits are required for an instruction for a 0, 1, 2 & 3 address machine.

3 Redo Example 2 for a machine with 255 instructions, 3 index registers, Indirect Addressing & the need to address up to 4194303 words.

CHAPTER 12

THE STRUCTURE OF AN ASSEMBLER

INTRODUCTION

The assembler program converts the code written in the assembler language into the Binary Machine Code of the target machine. The most common type of assembler program is the 2-pass assembler. 1-pass assemblers also exist. Both types will be described. Emphasis is placed on the 2-pass assembler. An example, done step by step, is given of the 2-pass assembly process. Finally the topics of Linking & Loading are discussed.

2-PASS ASSEMBLER

Two passes through the assembler program are made in order to produce the Binary Machine Code of the Target machine. The major activities of each of the passes is described below and shown in Figures 12.1 and 12.2 respectively.

PASS1

1. The Location-Counter value is managed.

The Location counter keeps track of the address, in Main Memory, of the instruction currently being assembled.

Initially the Location Counter is set to zero (This implies that the first instruction of the assembler program is placed in word 0 of Main Memory).

The Location Counter is incremented by 1 for each instruction as it is assembled. The exception is BSS instructions that can define any number of stores. In this case the Location Counter is incremented by the number of stores defined.

2. All LABELS are entered into a SYMBOL TABLE. Each Label together with its Location-counter value (which is the address of the Label) is entered into the Symbol table. Should a DUPLICATE Label occur an error message is generated.

3. Validates the Op-code mnemonics. It also provides suitable error messages for invalid op-codes. An error message is also given for missing or additional operands associated with an op-code.

4. It interprets Pseudo-operations completely.

PASS 2

The major activity of the second pass is to produce the Binary Machine code of the target computer. The SYMBOL TABLE set up in Pass 1 is used in conjunction with the ASSEMBLER CODE.

1. OPERAND FIELD is evaluated & inserted.
2. BINARY OP-CODE for the instruction is inserted.
3. ADDRESS MODIFICATION information is inserted into the instruction (ie Indirect or Indexed instructions).
4. An ERROR message is given for any IDENTIFIER whose name does not exist in the symbol table.

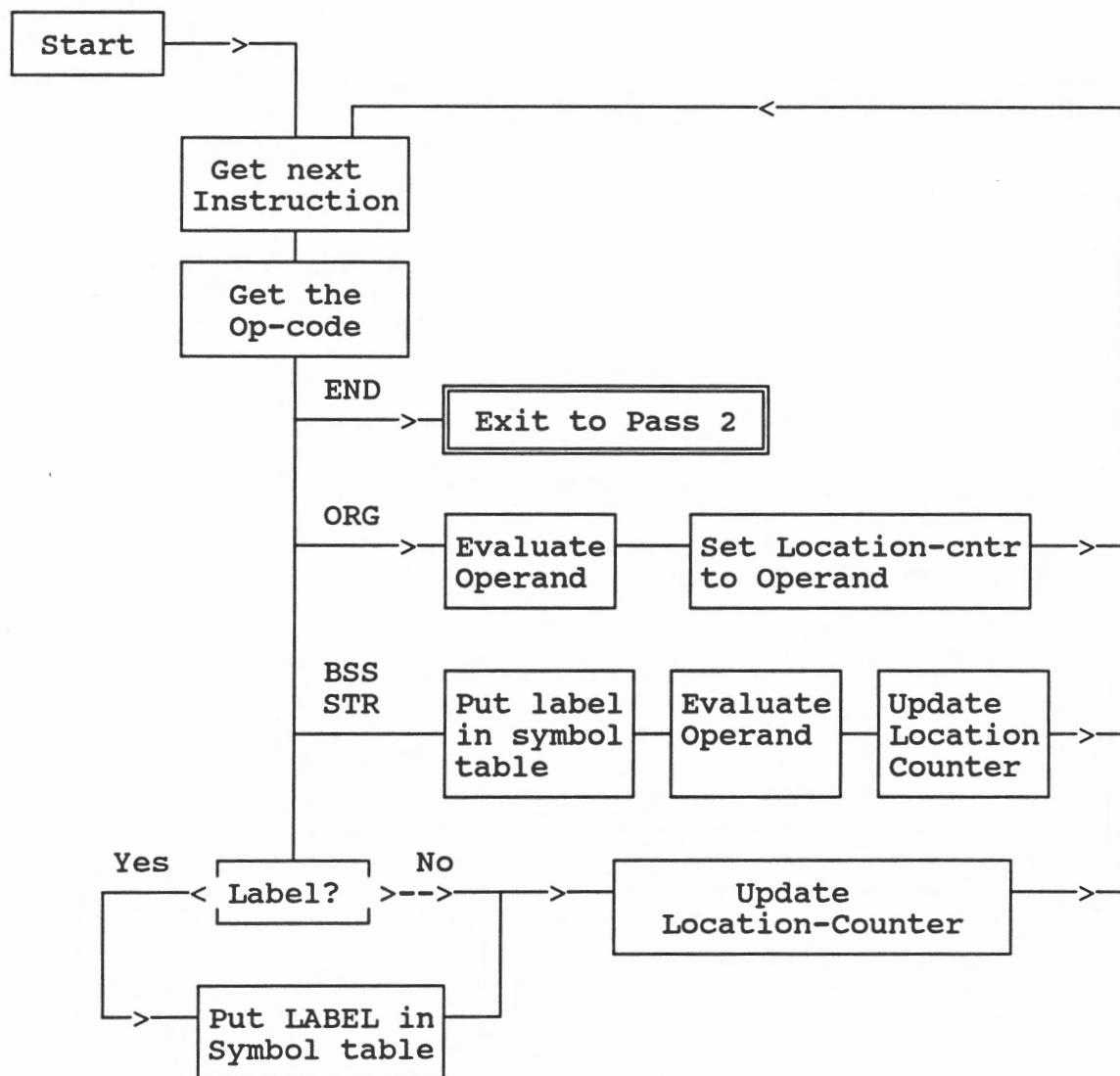


FIGURE 12.1 FLOWCHART OF PASS 1 OF THE ASSEMBLY PROCESS

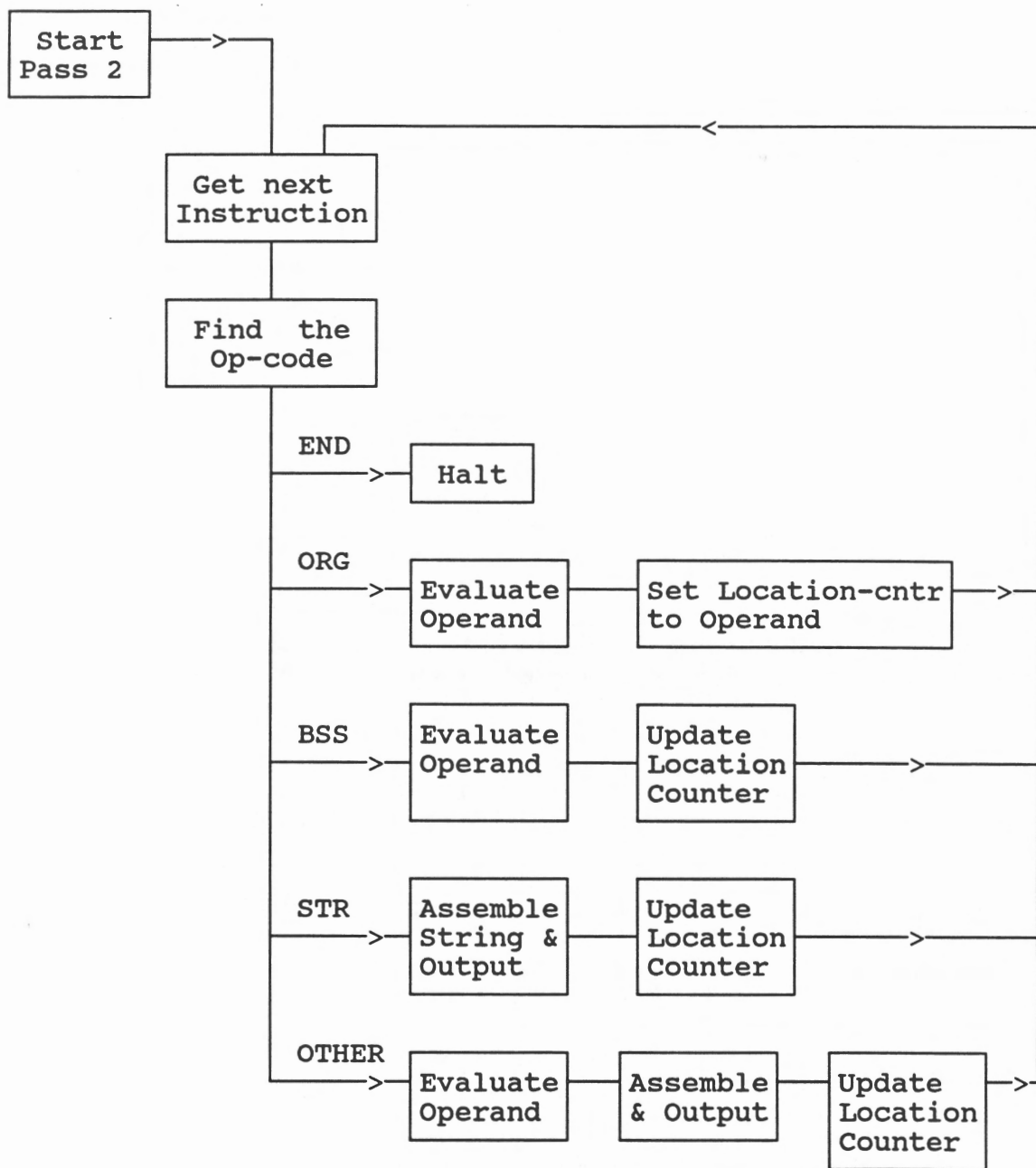


FIGURE 12.2 FLOWCHART OF PASS 2 OF THE ASSEMBLY PROCESS

NOTE: when an assembled statement is "output" the Binary Machine Code of that statement is placed in the store of Main Memory indicated by the Location Counter. This is known as a Load & Go Assembler.

EXAMPLE

The program used in the example has no real meaning. It is used for illustrative purposes only.

Location Counter		
0	start	INP
1		STORE a [0x]
2		JEQ label1
3	label2	ADD [aa]
4		SUB big
5		JGT label2
6	label1	JLT start
7		HALT
8	a	BSS 4
12	aa	ADDR a
13	big	DEC 4097
		END

During Pass 1 (a) the Location Counter will be set to the values shown in the Left-hand column as each statement is analysed, and (b) the symbol table is set up.

SYMBOL TABLE	
Symbol	Address
start	0
label2	3
label1	6
a	8
aa	12
big	13

The entries in the Symbol Table are made in the order in which the symbols occur.

Inserted during statement 0
Inserted during statement 3
etc.

During Pass 2 the Binary Machine Code is constructed & is shown below. The process starts at the first instruction and uses the symbol table to get the address of any symbol.

For SMALL the Binary Op. codes are: HALT(0), INC(1), ADD(2), STORE(3), LOAD(4), JMP(5), JEQ(6), JGT(7), JLT(8), SUB(9), INP(10), OUTP(11), CALL(14), RET(15).

ASSEMBLER
INSTRUCTIONSBINARY MACHINE
CODE

			Mem Addr	Op Code	Addr Modf	Operand Address	Comments
start	INP		0	1010	0000	00000000	
	STORE	a [ax]	1	0011	0100	00001000	Index Reg used
	JEQ	label1	2	1010	0000	00000110	
label2	ADD	[aa]	3	0010	0011	00001100	Indirect addr
	SUB	big	4	1001	0000	00001101	
	JGT	label2	5	0111	0000	00000011	
label1	JLT	start	6	1000	0000	00000000	
	HALT		7	0000	0000	00000000	
a	BSS	4	8	0000	0000	00000000	4 stores for
aa	ADDR	a	9	0000	0000	00000000	'a' defined.
big	DEC	4097	10	0000	0000	00000000	Initially they
	END		11	0000	0000	00000000	are all 0.
			12	0000	0000	00001000	aa=Address of a
			13	0001	0000	00000001	Value of 4097

ERROR MESSAGES

The SMALL assembler produces error messages wherever possible. During Pass 1 (a) it reports any duplicate name definitions and checks that all op. codes are valid; and (b) it reports on missing or excess operands (In some assemblers these errors are reported in Pass 2). During Pass 2 undeclared names are reported.

Example:

```

0      INP      a
1      STORE
2      SUB      three
3      JEQ      stop
4      AD       a
5      stop     ADD      a
6      OUTP
7      stop     HALT
8      a        BSS      1
9      thee     DEC       3
          END

```

On Pass 1 the following 4 errors are found:

Line 0: Parsing syntax error.(Extra operand).
 Line 1: Error. Operand expected.
 Line 4: Parsing syntax Error. (Invalid Op code).
 Line 7: Identifier declared more than once.

Assuming that these are corrected. Then on Pass 2 only 1 error will be found:

Line 2 : Identifier not declared

1-PASS ASSEMBLER

A 1-pass assembler does the same job as the 2-pass assembler but employs some clever data structures and strategies so that only 1-pass through the user program is necessary.

In essence : Each instruction, in turn, is converted to Binary Machine Code. A symbol table is kept as before. Any new symbol is added to the symbol table. If the address of this symbol is known then it is added to the table. If the address of a symbol can not be determined a linked list is created. The entry in the symbol table is now a pointer to the first occurrence of that symbol and so on. When that specific symbol's address is finally known (ie defined) the linked list is followed back and the address filled in wherever necessary.

EXAMPLE:

Several steps in the 1-pass assembly of the program to calculate $(a-2) + a + a$ is shown below.

```

      INP
      STORE a
      SUB   two
      ADD   a
      ADD   a
      OUTP
      HALT
a      BSS   1
two    DEC   2
      END

```

After statement 0 has been assembled:

Addr	Op Code	Addr Modf	Operand Address
0	1010	0000	00000000

INP

SYMBOL TABLE		
Symbol	List	Address

After statement 1 has been assembled:

	Addr	Op Code	Addr Modf	Operand Address	
INP	0	1010	0000	00000000	
STORE a	1	0011	0000	11111111	<

SYMBOL TABLE		
Symbol	List	Address
a	1	1

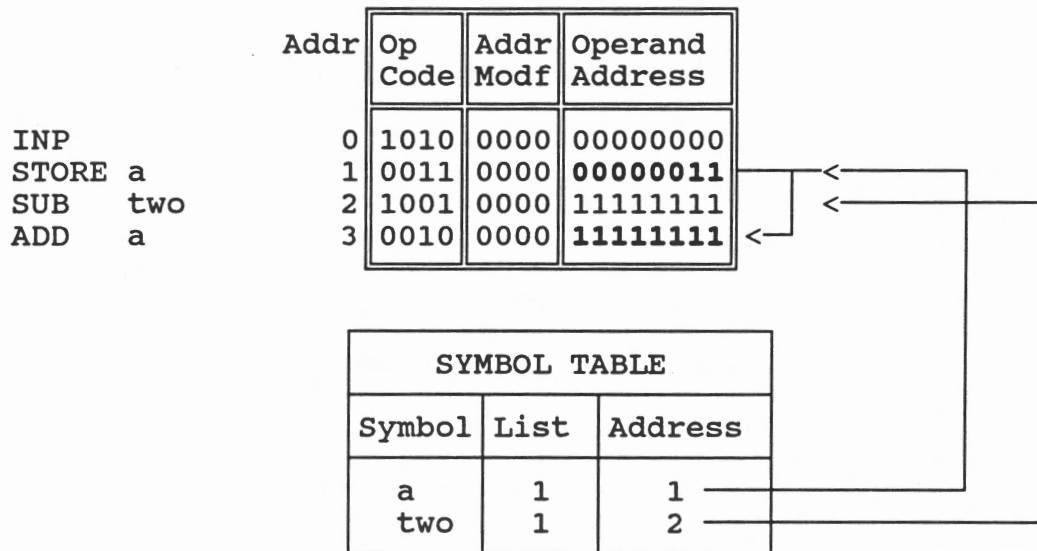
Note: A linked list is created for the address of 'a'. 'a' has not as yet been defined. When it is declared then only is this linked list used to fill in the actual address.

After statement 2 has been assembled:

	Addr	Op Code	Addr Modf	Operand Address	
INP	0	1010	0000	00000000	
STORE a	1	0011	0000	11111111	<
SUB two	2	1001	0000	11111111	<

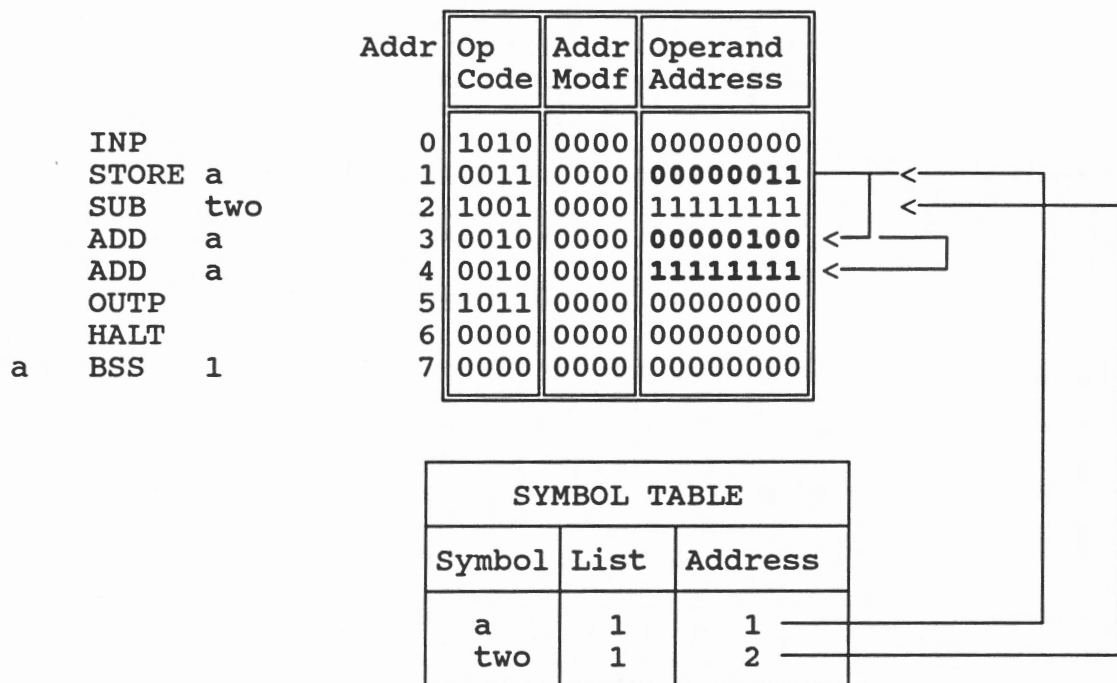
SYMBOL TABLE		
Symbol	List	Address
a	1	1
two	1	2

After statement 3 has been assembled:



After statement 7 has been assembled: The address of store 'a' is known to be 7. The linked list starting in the symbol table can be followed and the value 7 inserted in the address field of each such instruction.

BEFORE:



AFTER:



		Addr	Op Code	Addr Modf	Operand Address	
	INP	0	1010	0000	00000000	
	STORE a	1	0011	0000	00000111	
	SUB two	2	1001	0000	11111111	
	ADD a	3	0010	0000	00000111	
	ADD a	4	0010	0000	00000111	
	OUTP	5	1011	0000	00000000	
	HALT	6	0000	0000	00000000	
a	BSS 1	7	0000	0000	00000000	

SYMBOL TABLE		
Symbol	List	Address
a	0	7
two	1	2

SOME MISCELLANEOUS INTERESTING FACTS ABOUT ASSEMBLERS

1-PASS ASSEMBLERS

Should all the symbols be defined before they are used then there is no need to use the linked list concept to keep track of the unresolved addresses. The following technique, of defining all the stores first and jumping over them to the first executable instruction of the program can be used:

	JMP start	
a	BSS 4	;  Definition of all data
aa	ADDR a	;  Start of program.
ten	DEC 10	;
start	INP	;
	STORE a	
	
	etc	
	
	HALT	
	END	

Lets compare this with Pascal. Pascal is a 1-Pass Compiler. All data stores, procedure names etc have to be defined before they are used. The reason for this is that the symbol table can be set up and any symbol can quickly be found. No second pass, or linked list structure is needed to resolve any addressing issue.

THE FIRST ASSEMBLER

The first assembler had to be written in the BINARY MACHINE CODE of the machine for which it was to be used. This was a tedious and difficult job as you can imagine.

CROSS ASSEMBLERS

This is an assembler program that runs on machine 1 but produces the Binary Machine Code, not of that machine, but of some other machine, Machine 2 say.

This is a very useful concept. Once the FIRST Assembler has been written it can be used to write an Assembler program for any other machine. This can now be done at ASSEMBLER level. The only aspect of that first assembler that needs to be modified is the actual Binary Machine Code that it produces. It must produce the code for the new machine & not for the old one. This is a relatively easy change to make.

EXAMPLE: Lets assume that instructions in Machine 2 have the following format:

4 bit Op Code {LOAD is 1111 (ie different from Machine 1)}
 10 bit Operand address.
 2 bit Indexing {00 -None, 01 -- X-Index, 10 -- Y-Index.}

LOAD A [@x] would be cross assembled as follows:
 (Assume A=17 = 10001₂)

In Machine 2:

Op Code	Operand Address	Index
1111	0000010001	01

In SMALL:

Op Code	Index Mode	Operand Address
1000	0100	00010001

RELOCATION

In all the examples that have been done so far it has been assumed that the 1st instruction of the program is in word 0 of memory, the 2nd instruction in word 1 of memory and so on. This has been a valid assumption because the SMALL assembler has in fact done this. SMALL assembles the code and LOADS it to the memory of the computer. In addition the SMALL assembler always loads the assembled code starting at word 0 of the machine.

We will now study the implications of loading the program not starting at word 0 of memory but somewhere else. An example, shown in Figure 12.3, illustrates the differences when the program is loaded starting at word 16 rather than word 0.

ASSEMBLER				MEMORY OF COMPUTER			
<u>PROGRAM STARTS AT ADDRESS 0</u>							
				Addr	Op Code	Addr Modf	Operand Address
start	INP			0	1010	0000	00000000
	STORE	a		1	0011	0000	00000111
	ADD	five		2	0010	0000	00001010
	STORE	[bb]		3	0011	0011	00001001
	OUTP			4	1010	0000	00000000
	JEQ	start		5	0110	0000	00000000
	HALT			6	0000	0000	00000000
a	BSS	1		7	0000	0000	00000000
b	BSS	1		8	0000	0000	00000000
bb	ADDR	b		9	0000	0000	00001000
five	DEC	5		10	0000	0000	00000101
	END						
<u>PROGRAM RELOCATED TO START AT ADDRESS 16</u>							
				Addr	Op Code	Addr Modf	Operand Address
start	INP			16	1010	0000	00000000
	STORE	a		17	0011	0000	00010111
	ADD	five		18	0010	0000	00011010
	STORE	[bb]		19	0011	0011	00011001
	OUTP			20	1010	0000	00000000
	JEQ	start		21	0110	0000	00010000
	HALT			22	0000	0000	00000000
a	BSS	1		23	0000	0000	00000000
b	BSS	1		24	0000	0000	00000000
bb	ADDR	b		25	0000	0000	00011000
five	DEC	5		26	0000	0000	00000101
	END						

FIGURE 12.3

What is the difference between these two pieces of code?. In essence the new start address (16 in this case) has been added to the address part of **most** instructions. To which instructions is this new start address NOT added to? (1) To instructions that do not use the address part of the instruction (ie INP, OUTP, HALT); (2) To instructions that define data values, or stores, for use by the program (ie BSS, DEC, STR).

HOW RELOCATION IS DONE & HOW THE CODE IS LOADED TO MEMORY

Commonly the Assembler program produces 1 extra bit of information -- the **RELOCATION BIT**. This bit is set to 1 if the relocation address is to be added to this instruction else it is set to zero. The assembler has all the information necessary to decide whether the relocation address needs to be added or not.

Relocation of program code and the **LOADING** of that code to memory go hand in hand.

METHOD 1

The easiest way to relocate is for the assembler itself to do the relocation. This is fine if the relocation address is known at the time that the assembly is done. In **Small** this is the method used. **Small** can be considered to relocate the program to start in word 0 of memory. **Small** also automatically **LOADS** the assembled instruction to the appropriate word of memory. **Small** is known as a **LOAD & GO** assembler. This is because **SMALL** assembles the code, loads it to memory and automatically executes the program (well as soon as you press **RUN**).

METHOD 2

Often the relocation address is not known at assembly time. It frequently occurs that the user wishes to assemble a program, put this assembled code in a file and only later use it. In this case the Assembler will put the assembled code **which now includes the relocation bit** in the file. Later, and as a separate operation, that assembled code will be relocated and loaded to memory. The **RELOCATING LOADER** will do this job. The steps in this process are shown below:

```

1  START := RELOCATION START ADDRESS
2  RELOCATION COUNTER (RC) := START
   REPEAT
3    Get Instruction
4    IF relocation bit set then Add START to Instr.
5    STORE this instruction in word RC of memory.
6    RC := RC + 1
   UNTIL last instruction is relocated

```

METHOD 3

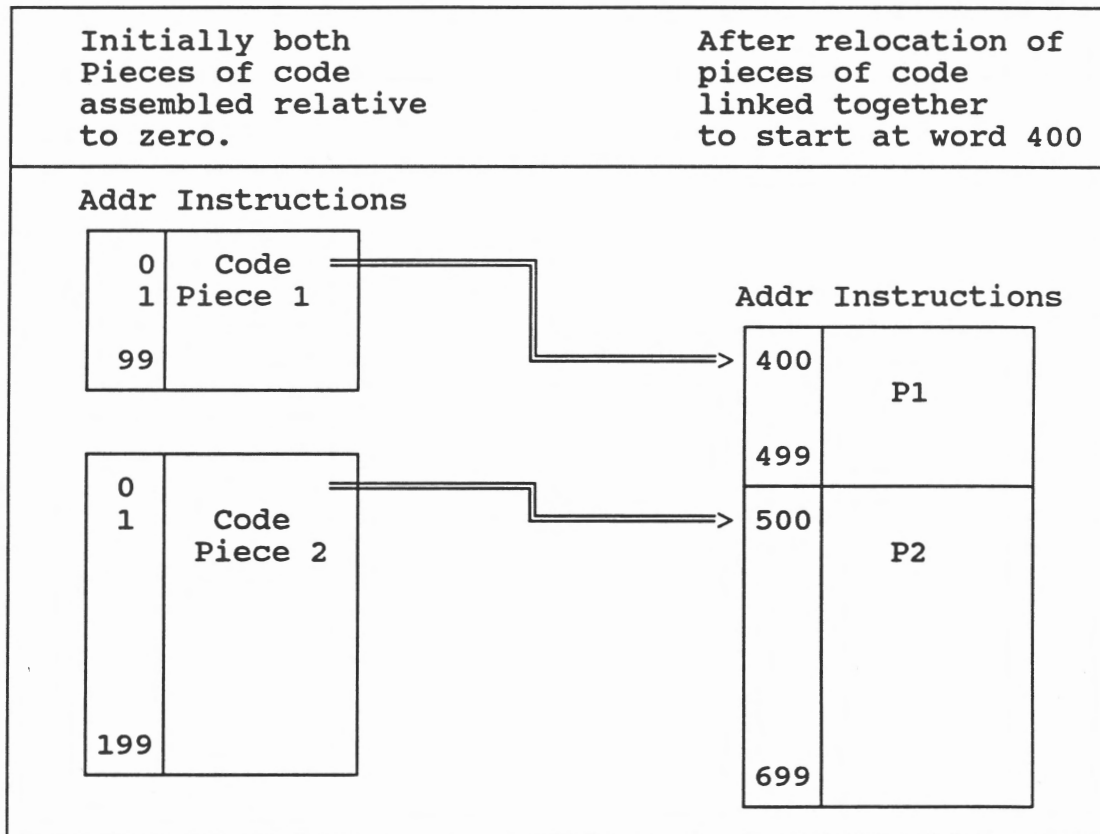
The assembled code, with addresses calculated relative to 0, is loaded to whatever relocation start address is required. The relocation bit has to be kept as part of each instruction. When this program is executed the **RELOCATION START ADDRESS** is loaded to a special CPU register, called the **RELOCATION (or BASE) Register**. As each instruction is executed the value in the **RELOCATION** register is automatically added to the instruction, if the Relocation bit is set.

This is a popular method to use. When a multiprogramming operating system is being used the Operating System can move such a program to some new place in memory. All the Operating System then needs to do is change the value in the **RELOCATION Register** (to the new start address of the program) for everything to work correctly.

LINKING

Let us consider the next step where we want to join two (or more) separately assembled pieces of code. There are two issues to consider. One is the relocation the other is how to deal with symbols defined in one piece of code but used in the other. A special linking loader is used that deals with both these issues.

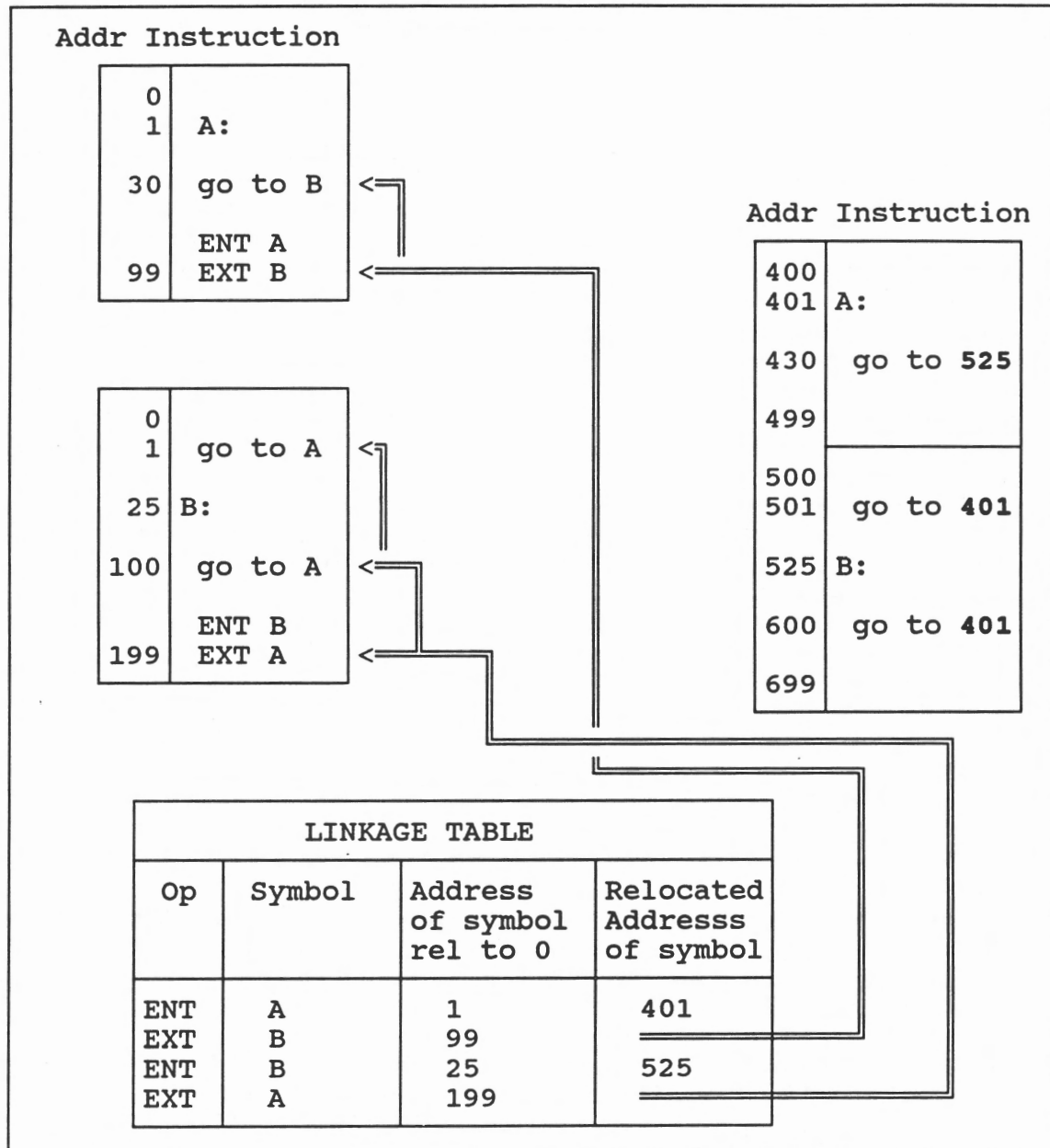
The relocation issue is reasonably easy. Assume that we have two pieces of code both assembled relative to zero. The first is relocated to the new start address. The second is relocated to the new start address + the length of the first piece of code.



The linkage of two pieces of code is the more difficult exercise. It is now possible in the first piece of code to use symbols(addresses) defined in the second piece of code and vica-versa. To allow for this two new pseudo-instructions are needed for external reference:

External	EXT	This marks a symbol as defined in another piece of code. ie EXT BCD
Entry	ENT	This marks a symbol defined in this piece of code that is used by another piece of code. ie ENT XYZ

When a sequence of pieces of code with external references is loaded, the loader places all ENT references in the symbol table, together with the addresses at which the ENT symbols are loaded. For each EXT symbol the loader finds the corresponding address in the symbol table and places this address in every instruction that references this symbol. (Casting our minds back to the 1-pass assembler the EXT symbol can be thought of as a link to the address(es) that have to be changed). A small illustrative example is given below:



The above description is intended to be sufficient for the reader to gain some insight into the problems of a relocating linking loader and to see a possible solution. I hasten to add that the illustration given here is not intended to be comprehensive. As an example of this what would happen if there are two EXT symbols with the same name?

CHAPTER 13

INPUT/OUTPUT CONCEPTS

In this section Input & Output is discussed. Initially the very simplest, and slowest, system is described. Interrupt driven I/O is introduced as is Direct Memory Access and the Channel concept.

PROGRAMMED INPUT/OUTPUT -- THE SIMPLEST SYSTEM

When a simple I/O instruction is to be executed: first the entire I/O operation is carried out; then only is the next instruction in the program sequence carried out.

This seems no different to what happens for any instruction so why remark on it? The answer to this is that any I/O instruction is a lot more complex to execute than for instance a Load instruction. This is because when a value is transmitted between the CPU and the I/O unit concerned it can take a variable length of time to complete. The instruction is deemed complete only when the I/O device concerned reports this fact back to the Control Unit by toggling the DATA flip-flop. Appreciate that all I/O is slow taking a large number of CPU cycles to complete. So many schemes are used to speed up I/O.

We now look at the I/O transfers in greater detail. First an Input Instruction and then an Output.

INPUT

Consider the SMALL instruction **INP**. A value is sent from the input device to the Accumulator. All the actions of this instruction are given below in Figure 13.1 & illustrated in Figure 13.2.

ASSM INSTR	CPU ACTION	INPUT DEVICE ACTION
INP	1 Clear Data flip-flop 2 Set Input line. 3 Test Data flip-flop Jump 3 if still clear. 5 Move Data from Data Buffer to Accumulator	4 Get data value & put it in the Data Buffer. When complete SET the flip-flop

FIGURE 13.1

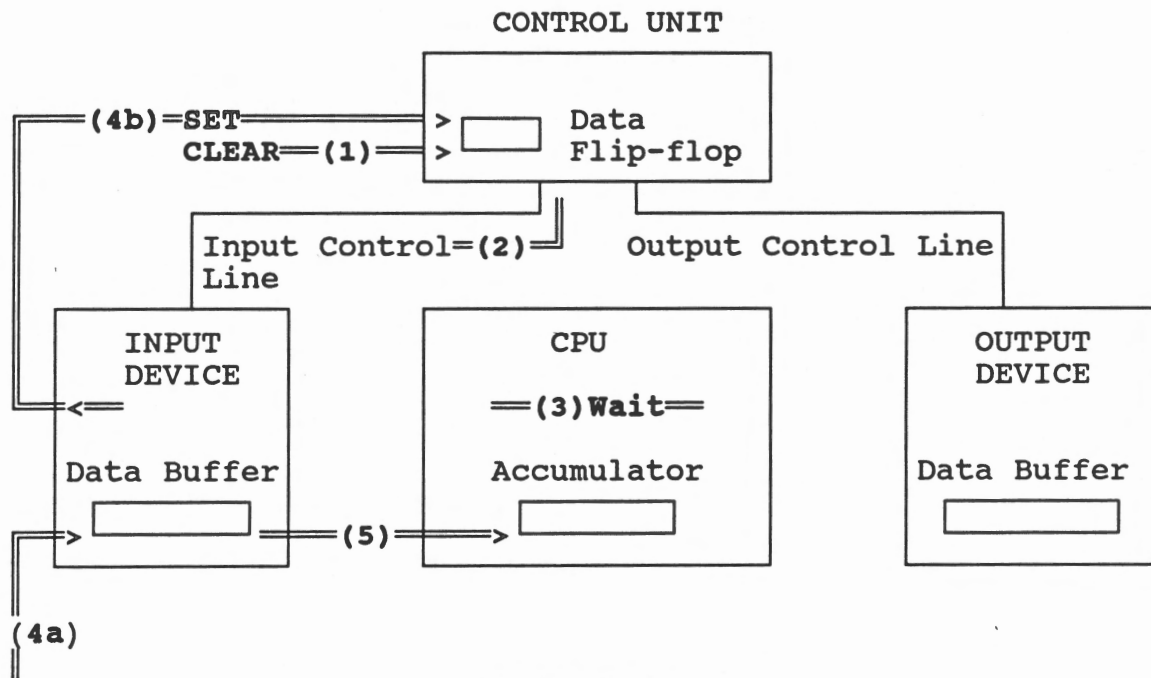


FIGURE 13.2

OUTPUT

Consider the SMALL instruction **OUTP**. A value is sent from the Accumulator to the output device. All the actions of this instruction are given below in Figure 13.3 & illustrated in Figure 13.4.

ASSM INSTR	CPU ACTION	OUTPUT DEVICE ACTION
OUTP	1 Set Data flip-flop 2 Set Output line. 3 Gates data from ACC to Data Output Line(DOL) 4 Test Data Flip-flop set. If yes Jump 4 6 CPU resumes program execution.	5 Output Device gates Data Output Line (DOL) to its Data Buffer CLEARS the Data flip-flop.

FIGURE 13.3

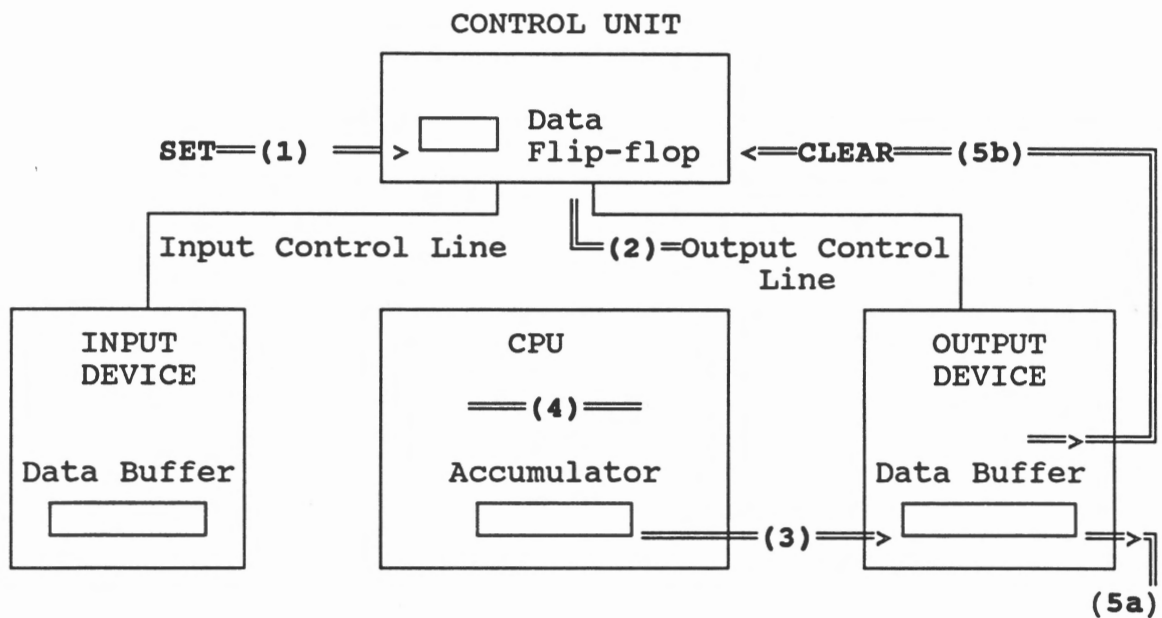


FIGURE 13.4

SPEED OF A SIMPLE I/O OPERATION

The speed of a simple I/O operation is very slow indeed. As an illustration the speed of the CPU and the speed of a typical printer will be compared. The values used in the example are typical for a common micro-computer.

1 memory cycle = 100 nanosecs = $100 * 10^{-9} = 10^{-7}$ secs.

Printer speed = 160 chars/sec.

Printing time for 1 char = $1/160 = .00625 = 6.25 * 10^{-3}$ secs

No. of memory cycles used (WASTED) while 1 char is printed

$$= 6.25 * 10^{-3} / 10^{-7} = 6.25 * 10^4 = 62500$$

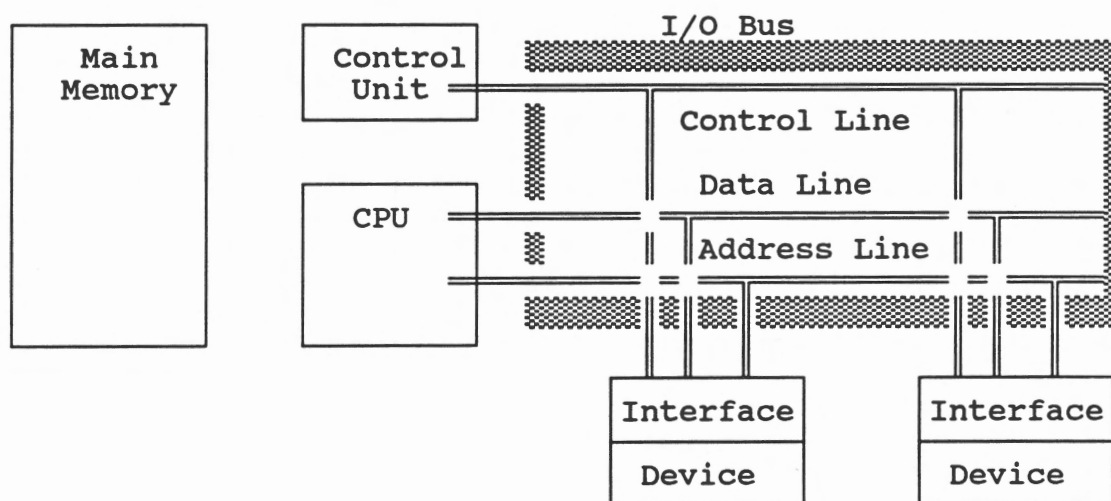
For 1 line of say 60 chars the WASTED memory cycles

$$= 62500 * 60 = 3750000 = 3.75 * 10^6 \quad (\text{Nearly 4 Million}).$$

It is easy to conclude that a much more efficient I/O system needs to be implemented. In essence we need to let the CPU do some useful computing while the Input or Output is being done in parallel.

MULTIPLE I/O DEVICES

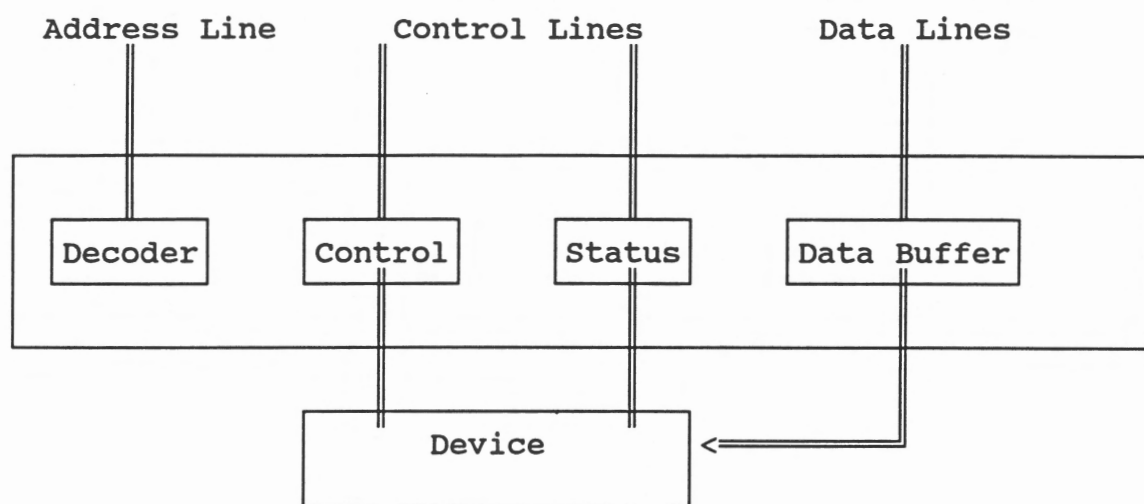
It is common for a computer to have several I/O devices attached to it. The principles used are the same as already described. An I/O bus is used to send all the signals (Control, Data & Address) to the devices. Because several devices are attached to the bus the Specific Device to be used must be indicated (Address {also known as the device number}).



The DATA line is used to transfer the data between the CPU & the device. The ADDRESS line is used to specify which device is to be used. The CONTROL line is used for timing and control pulses.

DEVICE INTERFACE

The device interface has several registers that are used to drive the device. These are shown in the diagram given below. Their function is then briefly described.



The **DECODER** takes the signal from the Address line and decides whether it applies to this device or not.

The **CONTROL** and **STATUS** circuits handle signals from both the CPU and the device relating to the functioning of those units.

The **DATA BUFFER** is a buffer to hold the data being transferred between the CPU and the device.

ERROR DETECTION and **CORRECTION** is also performed in the Device interface on the data in the Data Buffer. **DATA CONVERSION** between different representations of data can also be performed on the data in the Data Buffer.

MORE SOPHISTICATED I/O STRUCTURES

In this section Interrupt driven I/O will be discussed. Direct Memory Access (DMA) or the use of **CHANNELS** will also be described.

These methods are effectively used only when a Multiprogramming Operating System is used. Under control of such an Operating System control can be swopped from one user program to another. Effectively while one user is waiting for I/O to be done the unused CPU cycles can be used by some other user to do useful computations.

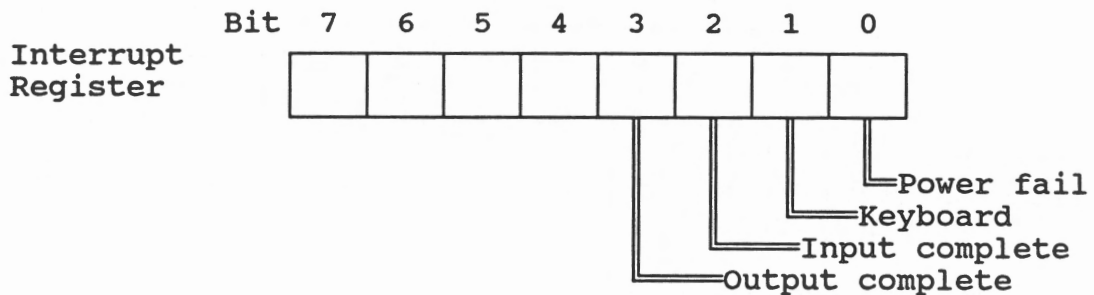
For single user operating systems these sophisticated schemes give little advantage because the single user can do little (or no) useful other computing until the I/O transfer has been completed.

INTERRUPT I/O SYSTEM

Instead of the CPU endlessly checking the Data flip-flop to see if the I/O unit has finished. Rather let us do things a different way. The I/O instruction is executed and it starts up the I/O operation. This operation is allowed to run on in parallel. When it is finished it INTERRUPTS the CPU. In the mean-time the CPU can be used to do useful computations (for some other user).

INTERRUPT HARDWARE

To effect such a system an INTERRUPT system must exist in the computer. A hardware interrupt register in the CPU is required. Each bit of this register indicates that a specific event has taken place. Together with this Interrupt register an interrupt system must exist that allows each device to set its bit in the Interrupt register at any time without in any other way affecting the action of the computer.



INTERRUPTS AND THE CONTROL UNIT

The action of the Control unit must be modified. At the end of each Execute cycle the Interrupt register must be checked. If an interrupt has occurred then it must be serviced. The Execute Phase of the Control Unit action is modified as follows:

FETCH	:
INDIRECT	:
EXECUTE	:	E1: Execute Instruction E2: P --> P+1 except for Jump Instructions E3: Service the Interrupt if one exists E4: Go to FETCH phase

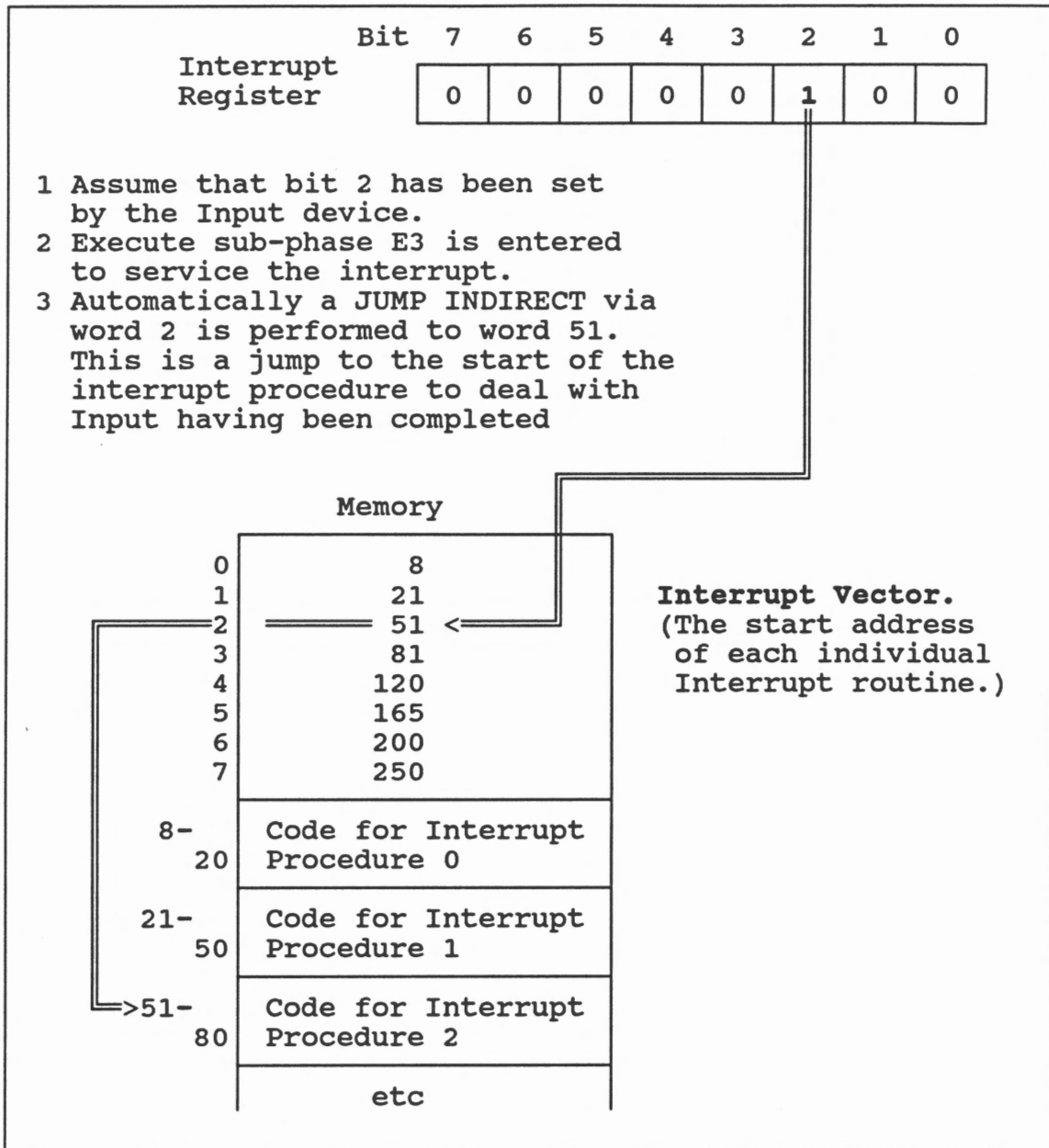
SERVICING AN INTERRUPT

The servicing of the interrupt is broken down into 2 parts:

- E3.1: Save the P-Reg value.
- E3.2: Jump to the start of the appropriate Interrupt procedure that deals with this interrupt.

In association with step E3.2 the first 8 words of memory (ie stores 0--8) are used as an Interrupt vector storing the start address of the appropriate Interrupt routine to service that interrupt. So if bit 1 is set a Jump indirect via word 1 is done (to word 8). If bit 2 is set a Jump indirect via word 2 is done and so on.

Example: All the steps in servicing an Input completed interrupt will be described.



DETAILED STRUCTURE OF AN INTERRUPT ROUTINE

An interrupt routine uses the CPU registers thus it must first store the values in the CPU registers away, do its processing then restore the CPU register values so that the original processing can continue.

The detailed structure of the Interrupt procedure is given below;

- 1 Disable the Interrupt System.
- 2 Save the processor status (ie All the CPU regs).
- 3 Enable the Interrupt System.
- 4 Service the Interrupt.
- 5 Disable the Interrupt System.
- 6 Restore the processor status (All the CPU regs+P-Reg)
- 7 Clear appropriate bit in Interrupt Register.
- 8 Enable the Interrupt System.

Should several interrupts be indicated at the same time they are dealt with in order of priority. Interrupt 0 first, 1 next & so on. This ordering is implemented by the interrupt hardware.

A MULTIPLE INTERRUPT EXAMPLE

In this example, shown in Figure 13.5, we assume a Multiprogramming Operating System. The same interrupt principles described above will be used. There are 3 user programs running in the system. In such a system at the end of an interrupt routine there is always a jump back to the Operating System for it to decide which user program is to run next.

In this example the I/O can be usefully overlapped. While one program is waiting for I/O the CPU can be usefully used for another program.

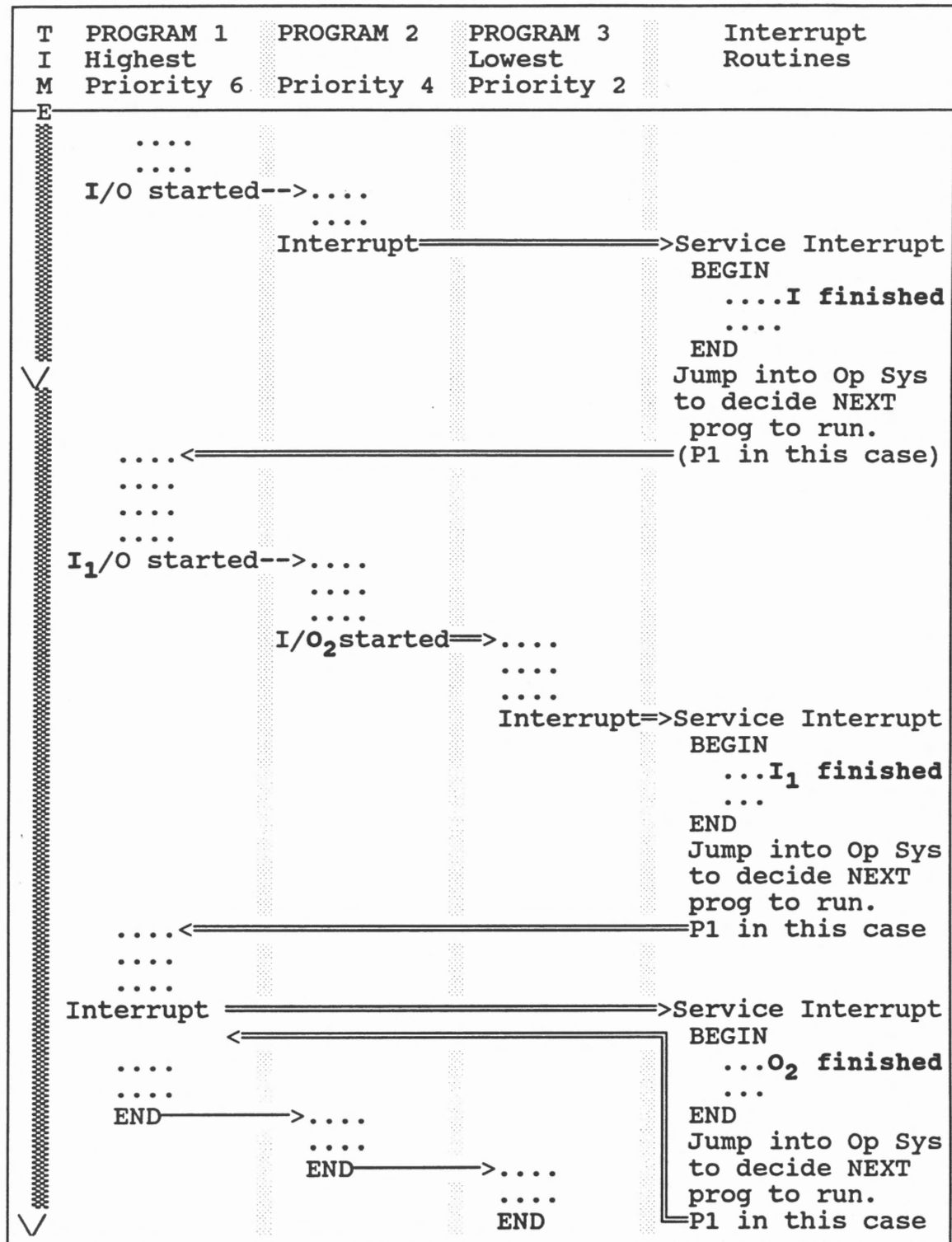


FIGURE 13.5

DIRECT MEMORY ACCESS CONTROLLER (DMA)

The Interrupt system scheme discussed above is a distinct improvement on the simple I/O model. This system does however still involve a fair amount of CPU intervention when a large amount of I/O is to be done. Let us consider the input of say 60 characters. This Input operation is started. After each character is obtained the CPU is interrupted to accept this character & put it somewhere. This is time consuming because not only does the specific character have to be dealt with but the CPU register values have to be stored & retrieved each time.

One solution is to add a special **ADDITIONAL** processor that controls the entire I/O transfer. This processor (the Direct Memory Access Controller) runs in parallel with the CPU.

The DMA is a limited capability processor, having its own controller, a Word Count Register, an Address Register & a Data Buffer & is shown in Figure 13.6.

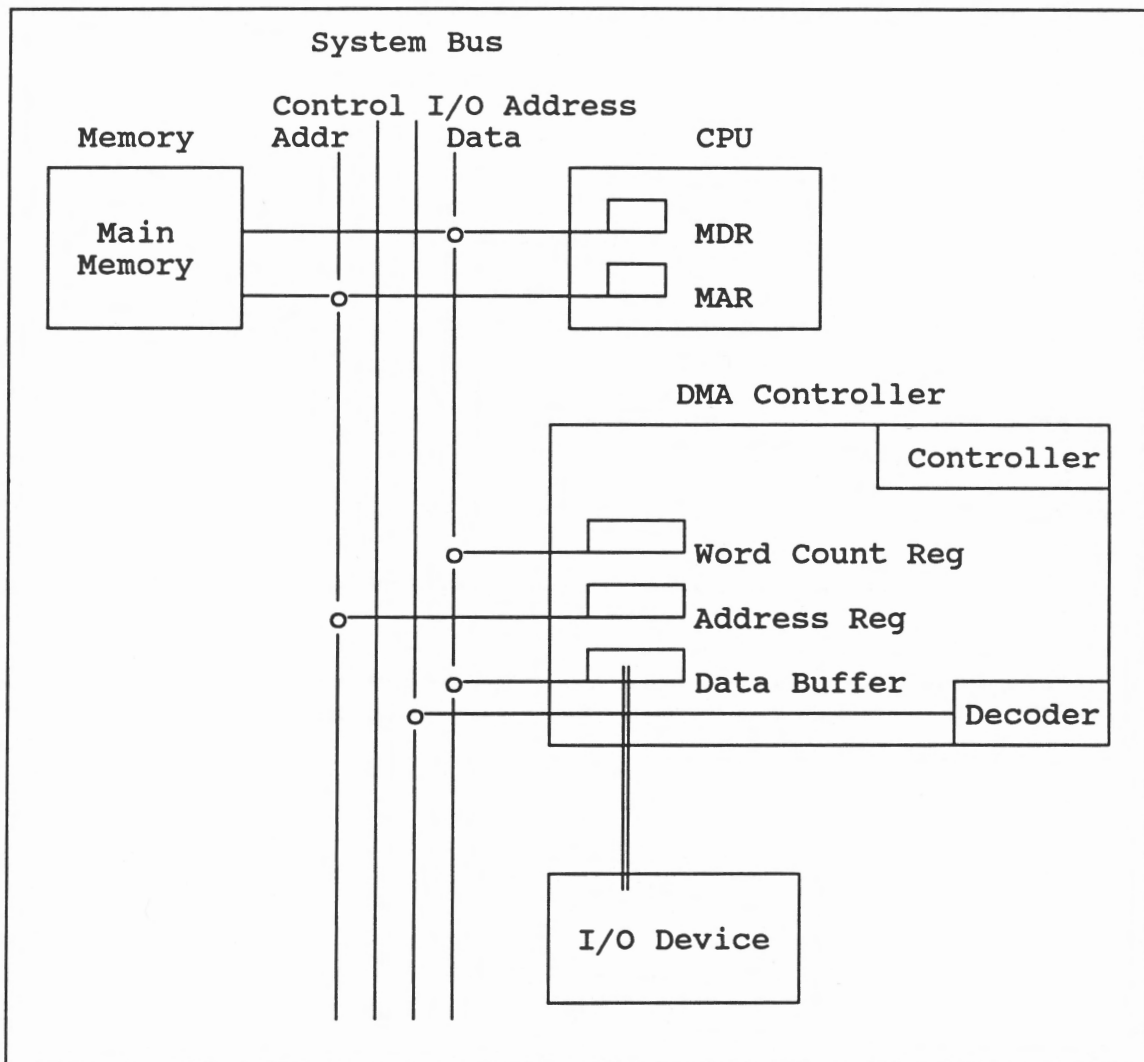


FIGURE 13.6

When an I/O operation is specified the appropriate values are loaded to the registers of the DMA.(ie Input or Output, the number of words to be transferred, the Address in Memory of these words). The DMA then performs the ENTIRE I/O operation. The CPU is only interrupted to steal a cycle when the DMA needs to access the Main Memory of the computer. These actions will now be detailed.

I/O INITIATION

The CPU initiates the I/O operation by:

- 1 Sending all the appropriate information to the DMA controller. (ie The number of Characters to be transferred, the I/O device to be used & the location in Main Memory for the data.
2. Sets the DMA processor running.
- 3 Transfers control to the Operating System so that some other process can use the CPU while the I/O operation is being done.

DMA CONTROLLER

For each character the DMA controller:

- (1) Gets the character from the I/O device
- (2) Steals a CPU cycle & transfers the Character to Main memory.

When the entire transfer is complete the DMA processor sends an interrupt to the CPU to indicate that the I/O operation has been completed. The CPU, on receipt of this interrupt, will service it. The Operating System will then decide which process is to continue running in the CPU.

CYCLE STEALING

When the DMA processor wishes to access Main Memory, either to send a character or to get a character, the CPU must be stopped from using Main memory at the same time. This is called CYCLE STEALING. The DMA processor issues a special interrupt that freezes the CPU for 1 cycle. During that cycle the DMA processor uses its Registers AR (for Memory Address) & the Data Buffer (for Data) to communicate with Main Memory.

CHANNELS

A CHANNEL is a sophisticated DMA processor. It can interface with several I/O devices at any one time rather than only one. It can perform extensive error detection & correction, data formatting & code conversion. It can also interrupt the CPU under certain I/O error conditions.

There are two types of Channels: Selector Channels & Multiplexor channels.

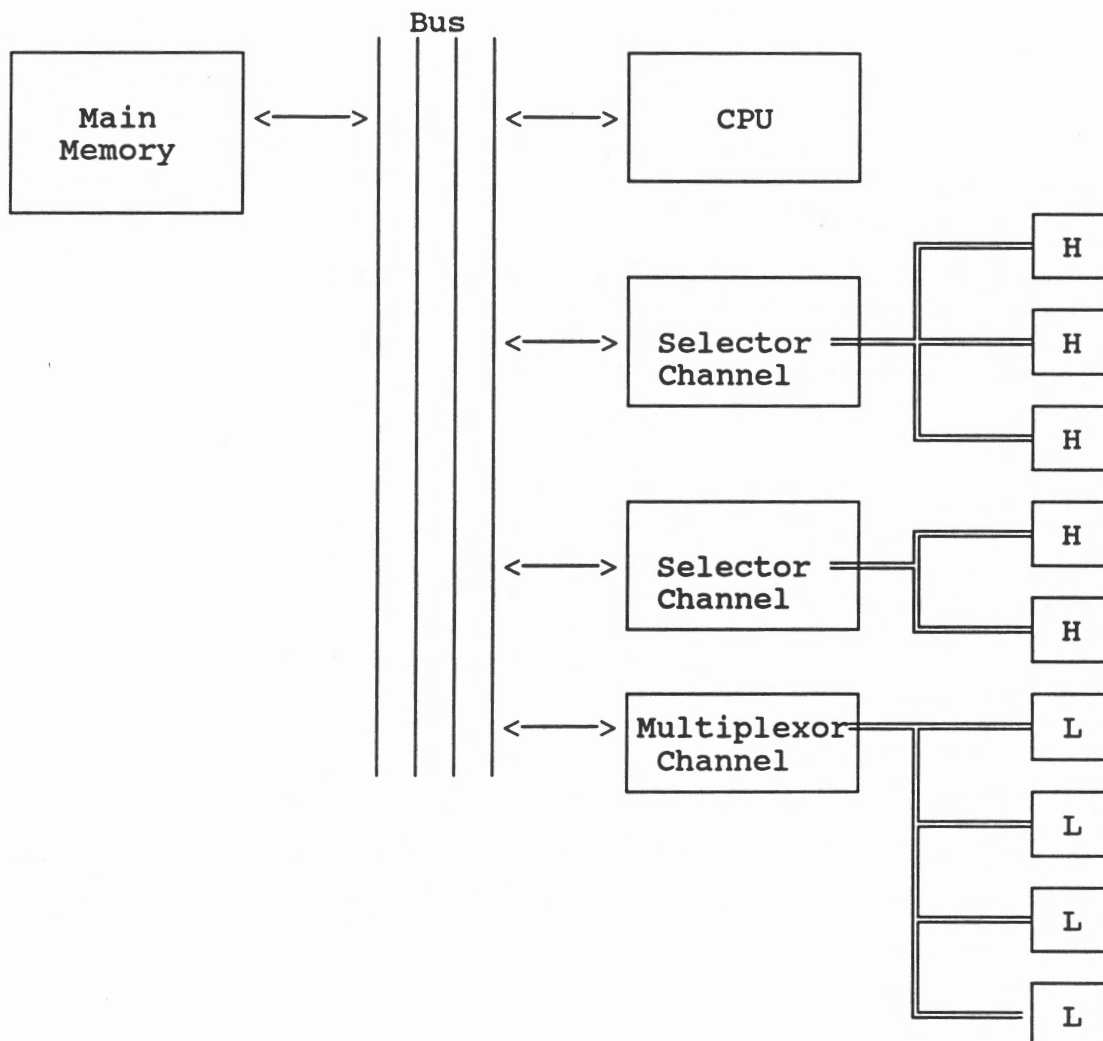
MULTIPLEXOR CHANNEL

This type of Channel is connected to SEVERAL slow/medium speed I/O devices. The Channel scans each device in turn, collects data into a buffer, steals a cycle & transfers the data to the Main Memory location appropriate to the device (or vice-versa). The transfer of data can either be 1 character or a block of characters. It should be emphasised that 1 Character (or block) is transferred from the first device, then 1 character (or block) from the 2nd device & so on in round robin style before returning to the 1st device again.

SELECTOR CHANNEL

This type of Channel is used for connection to high speed devices like tapes & disks. These high speed devices tend to keep the channel busy because of the high data transfer rates. Although several devices are connected to a selector channel, the channel stays connected to one device until the entire data transfer is completed. Only then will another device be selected.

A typical internal configuration of a large computer could be:



CHAPTER 14

CONTROL UNIT STRUCTURES

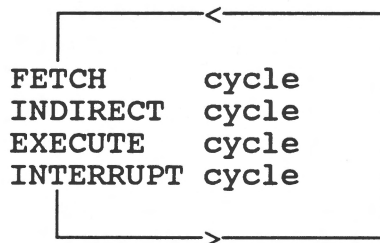
Hardwired and micro-programmed control unit structures will be described in this chapter. The attributes of both will be discussed.

STRUCTURE & FUNCTIONS OF THE CONTROL UNIT (CU)

We have already seen that the Control Unit contains the following Registers:

DATA flip-flop
 RUN flip-flop
 STATE register
 INTERRUPT register

The CU REPEATEDLY EXECUTES:



At each stage of each cycle the CU generates CONTROL SIGNALS which drive the rest of the computer to perform the required action.

Example: During the FETCH cycle the CU generates signals that :

```

1    P --> MAR
2    Memory Fetch: M[MAR] --> MDR
3    MDR --> I
```

During the EXECUTE phase the CU uses the contents of the I-Register as input to its circuitry which then generates the appropriate signals for the computer to execute that specific command. Take the STORE X instruction for example:

```

1    I7-0 --> MAR
2    ACC --> MDR
3    Memory Fetch: MDR --> M[MAR]
4    P --> P+1
```

Naturally a different sequence of signals is produced for each command.

TYPES OF CONTROL UNITS

There are 2 types: Hard-wired and Micro-programmed. Each will be described.

HARD-WIRED CONTROL UNIT

As the name implies the CU is made up logic circuits comprised of gates and flip-flops which are physically hardwired together. These circuits then generate the appropriate output (control) pulses that drive the action of the rest of the computer.

Once the hard-wired control Unit is designed and built it can not be changed. No instruction can be modified nor can new, extra, instructions be added.

MICROPROGRAMMED CONTROL UNIT

In the previous model a hardwired circuit is used to produce the control pulses at each sub-step of each cycle. Now a sequence of micro-instructions, one for each sub-step of each cycle is used to generate the control pulses. These micro-instructions are stored in a special, very fast, CONTROL READ ONLY MEMORY (CROM). The execution of this micro-program sequence is controlled by a simple hard-wired Micro-Control Unit that repetitively loads and executes micro-instructions. In addition to its other registers the CU has a micro-MAR, a micro-MDR, a micro-P & a micro-I register. All these register have exactly the same role as their counter parts have in the CPU except that they interconnect with the Control Read Only Memory and not the Main Memory. The micro-Instruction register now produces the control pulses that control the action of the rest of the computer. A diagram of a micro-programmable Control Unit is given in Figure 14.1. There-after some details of the micro-instructions are given.

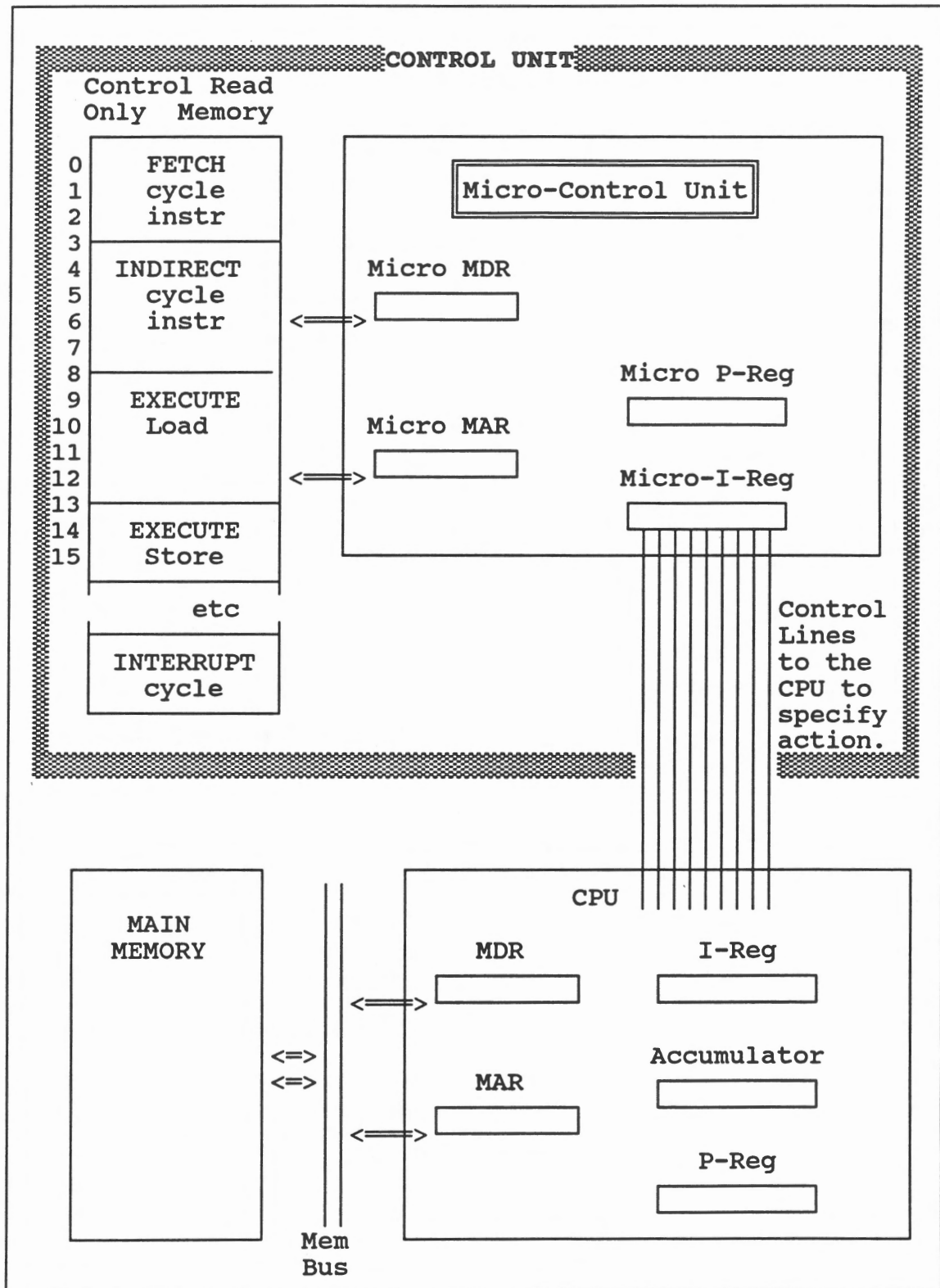


FIGURE 14.1

In detail, the micro-instructions could be: (The Indirect cycle is omitted)

FETCH:

- 1 P --> MAR
- 2 Memory Fetch: M[MAR] --> MDR
- 3 MDR --> I
- 4 Jump to start of EXECUTE specific Op Code
(Bits 15--12 of I-Reg used as jump address.
usually indirectly via a jump table).

EXECUTE (Store X):

- 1 I₇₋₀ --> MAR
- 2 ACC --> MDR
- 3 Memory Fetch: MDR --> M[MAR]
- 4 P --> P+1
- 5 Jump to start of INTERRUPT cycle

INTERRUPT:

- 1 Service interrupt
- 2 Jump to start of FETCH cycle

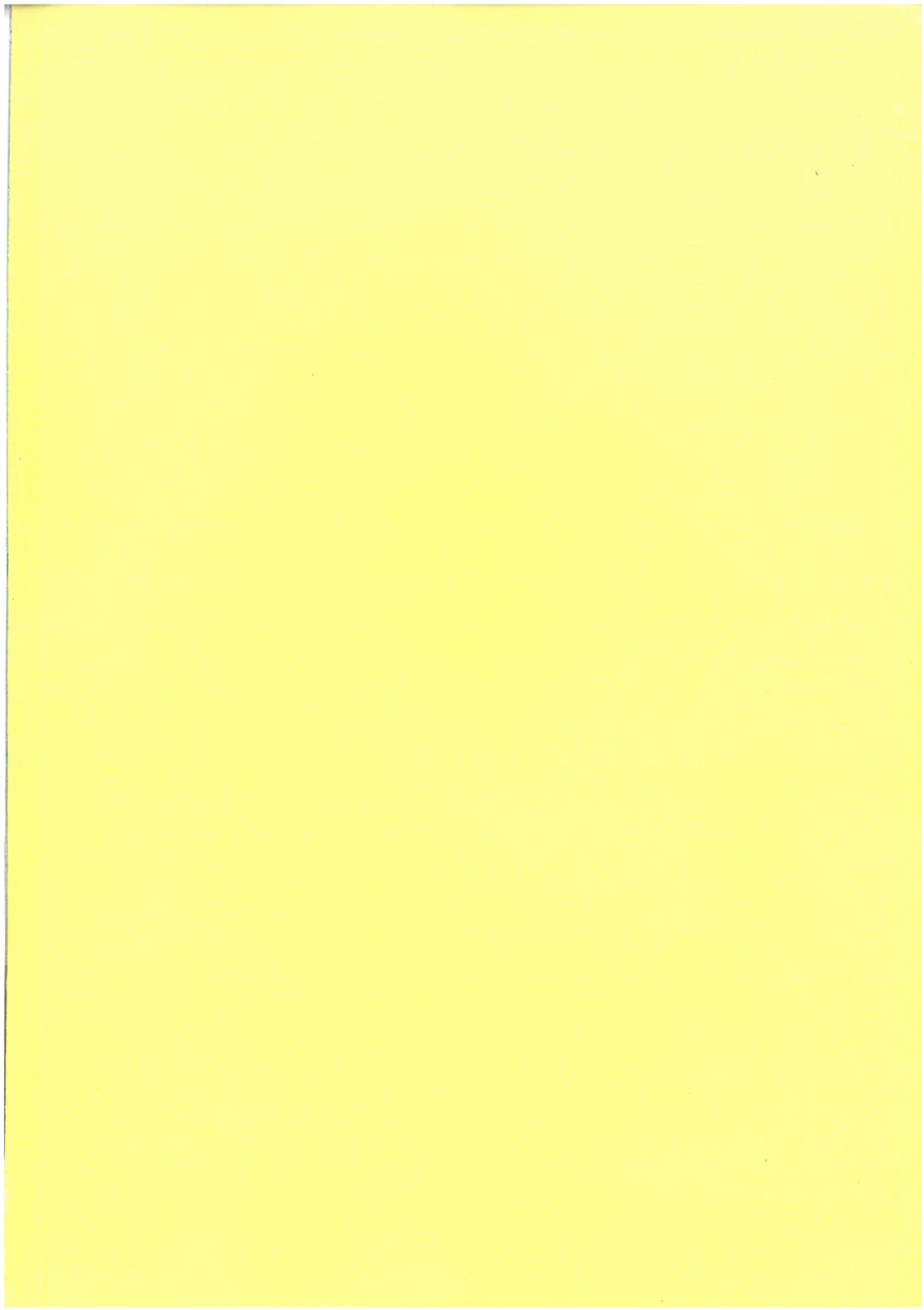
An assembler instruction can be considered to be a Macro-instruction made up of a series of micro-instructions. This series of micro-instructions is known as a micro-program. Note that a micro-program can not be interrupted. Interrupts are only recognised and serviced in the Interrupt cycle (as shown above).

ADVANTAGES OF THE MICRO-PROGRAMMED CONTROL UNIT

Appreciate that it is possible to change the micro-instructions in the CROM hence:

1. During the computer design phase it is easy to CORRECT ERRORS in instructions and to ADD new instructions.
2. The user can add new assembler instructions to the machine. This gives the advantages of creating special purpose assembler instructions. These instructions can in turn give the user greater ease of programming as well as being very fast to execute.
3. EMULATION of one computer on another. ALL the instructions of the computer can be re-written so that your computer looks & reacts exactly like some other machine. Manufacturers often use this facility when they wish to achieve COMPATIBILITY for a range of computers.

For most machines changes the micro-code can only be made by the manufacturer. If a user wants this facility then a suitable machine has to be purchased.



Q 1101101101

A 111101100

↓ The funny bit